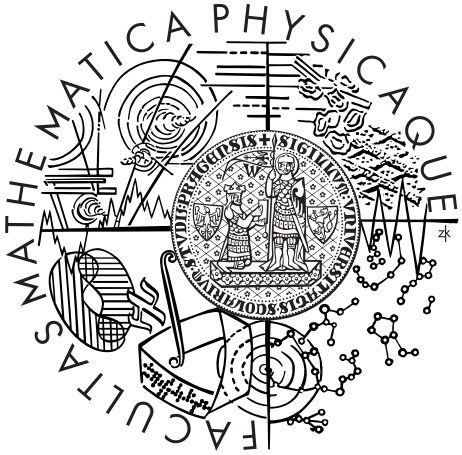
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Bc. Martin Podloucký

Automated GUI Generation for Functional Data Structures

Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce:	Ing. Robert Pergl, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika

Praha 2012

Děkuji vedoucímu práce Ing. Robertu Perglovi, Ph.D. za velmi inspirativní a motivující vedení stejně jako za jeho cenné rady a připomínky během mojí práce na tomto projektu.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V Praze dne 10. dubna 2012 Podpis autora

Název práce: Automated GUI Generation for Functional Data Structures

Autor: Bc. Martin Podloucký

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Ing. Robert Pergl, Ph.D., Katedra softwarového inženýrství, Fakulta informačních technologií, České vysoké učení technické v Praze.

Abstrakt: Tato práce se zabývá automatickým generováním grafického uživatelského rozhraní pro funkcionální programy. Po rozboru a zhodnocení současných možností v oblasti automatického generování GUI je představen koncept takzvaného funkcionálně strukturovaného uživatelského rozhraní (FSUI). Je specifikován systém metadat pro anotaci kódu v jazyce Clojure a popsána a implementována transformace z tohoto systému do datového modelu FSUI. Poté je v jazyce Clojure implementována grafická vrstva, která zobrazuje skutečné grafické rozhraní. Funkčnost tohoto přístupu je demonstrována na případové studii.

Klíčová slova: funkcionální programování, automatické generování GUI, Clojure

Title: Automated GUI Generation for Functional Data Structures

Author: Bc. Martin Podloucký

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Ing. Robert Pergl, Ph.D., Department of Software Engineering, Faculty of Information Technology, Czech Technical University in Prague.

Abstract: This thesis addresses the problem of automated graphical user interface generation for functional programs. First an analysis of current state in the field of automated GUI generation is performed. Based on the analysis the concept of Functionally Structured User Interface (FSUI) is introduced. Meta-data system for code annotation is then specified for the Clojure programming language and a transformation from this system to FSUI data model is implemented. Finally a graphical layer for displaying the actual interface is implemented in Clojure. Benefits of this approach are demonstrated by proof-of-concept case study.

Keywords: functional programming, automated GUI generation, Clojure

Obsah

Část I

Úvod

Kapitola 1

Motivace

Softwarové inženýrství je odvětví, jež neustále kráčí kupředu nepřetržitě hledajíc lepší techniky a technologie, které by umožnily vyrábět software rychleji, s menšími náklady a lepší výslednou kvalitou [1]. Velký tlak na inovace lze pozorovat především v oblasti business aplikací, při jejichž tvorbě je ekonomičnost vývoje klíčovým faktorem [2]. Právě toto je doména, ve které lze často s úspěchem využít toho, že business aplikace mají jisté společné charakteristiky. Mnohdy fungují na podobných principech, pracují s podobnými daty a musí se vypořádat s podobnými omezeními [2]. Výše popsaná skutečnost vede mnoho vývojářů k využívání technik takzvaného *generativního programování* [3].

Základ generativního programování spočívá v možnosti vygenerovat počítačový program automaticky z popisu zapsaného v jazyce, který je na vyšší úrovni abstrakce než jazyk zdrojového kódu výsledného programu. To například zahrnuje postupy, kdy výstupem generátoru je zdrojový kód aplikace v nějakém vysokoúrovňovém jazyce, jako jsou například Java, C++, Smalltalk a další. Vstupem pro takový generátor je pak nějaký doménově specifický jazyk pro popis chování aplikací a systémů, což může být například UML, BPMN a mnoho jiných. V generativním programováno nacházejí široké uplatnění techniky Model Driven Software Development (MDSD) [4], které umožňují z daného modelu vygenerovat i celou funkční aplikaci včetně grafického uživatelského rozhraní bez nutnosti psát ručně zdrojový kód. Tento přístup se často označuje jako Model Driven Architecture (MDA) [5].

1.1 Generování GUI z kódu

Dílčím a zajímavým problémem automatického generování software je generování grafického uživatelského rozhraní z již existujícího kódu aplikace. Tento problém vychází z myšlenky, že při vývoji aplikací je žádoucí oddělit vývoj logiky aplikace od tvorby jejího uživatelského rozhraní, neboť se tak snižuje provázanost jednotlivých částí programu [7]. Vývoj je pak efektivnější a program je snadněji udržovatelný [6]. V ideálním případě by tedy vývojáři napsali business logiku a až ve chvílí, kdy je funkční a otestovaná, by nad ní vytvořili GUI. Takový scénář se ale často nedaří dodržet ať už kvůli technickým omezením nebo proto, že softwarová firma chce zákazníkovi co nejdřív předvést alespoň hrubý obrys dodávaného softwaru [1], [2]. Programátoři jsou tedy často nuceni vyvíjet GUI spolu s logikou aplikace, i když je to často odvádí a zdržuje od v počátcích projektu podstatnějšího implementování základních funkcí [7]. Pokud je však možné vygenerovat alespoň provizorní uživatelské rozhraní přímo ze zdrojového kódu aplikace, toto zdržování může úplně odpadnout.

Při generování grafického rozhraní přímo z kódu však vyvstává zásadní problém. Na MDA založený generátor, který vytváří kód celé aplikace, tedy uživatelské rozhraní i business logiku, má na svém vstupu popis fungování celé této aplikace v nějakým modelovacím jazyce, například UML. Rozumí tedy tomu, co tato aplikace dělá. Naproti tomu generátor, který generuje uživatelské rozhraní přímo ze zdrojových kódů, musí logice aplikace porozumět jejich analýzou, neboť nic jiného než zdrojové kódy nemá k dispozici. To může být obzvláště pro některé druhy programovacích jazyků velmi netriviální úloha. Pro takové jazyky pak může nastat potřeba přidávat do kódu relativně velké množství anotací, aby byl generátor jednak schopen porozumět jeho struktuře a jednak byl schopen se vyrovnat s funkcionalitou specifickou pro tu kterou aplikaci. Množství anotací ve zdrojovém kódu pak znesnadňuje jeho čitelnost a udržovatelnost.

1.2 Funkcionální jazyky

Vzhledem k nutnosti provádět statickou analýzu kódu je výhodnější zkusit výše nastíněný problém vyřešit pro funkcionální jazyky. Těmi se v této prácí myslí

programovací jazyky, jako jsou Lisp, Haskell, Erlang, ML a další. Jazyky z této kategorie mívají jednodušší syntaxi než imperativní jazyky a používají alespoň většinou čisté funkce a neměnitelné datové struktury [8]. Jsou tak vhodnější pro statickou analýzu kódu, které bude při generování GUI třeba. Navíc v současnosti vznikají nové funkcionální jazyky, jakým je například Clojure, jejichž ambicí je použitelnost nejen v různých odvětvích matematiky a umělé inteligence, nýbrž také ve světě business a enterprise aplikací [28].

Kapitola 2

Cíl a metodika práce

Cílem této práce je přispět k řešení problému automatického generování grafického uživatelského rozhraní z anotovaného zdrojového kódu aplikace v souladu s motivací uvedenou výše. To zahrnuje

- Provést analýzu současného stavu v oblasti automatického, případně poloautomatického generování grafického uživatelského rozhraní.
- Navrhnout framework podporující automatické či poloautomatické generování abstraktního GUI pro funkcionální datové struktury
- Vytvořit transformaci z abstraktního popisu GUI do některého běžně používaného GUI frameworku.
- Implementovat případovou studii demonstrující výsledky práce.

2.1 Metodika

Následující body popisují metodiku použitou v této práci. Ukazují následnost kroků, které povedou k dosažení cíle vytyčeného v předchozí části.

1. Analýza současného stavu v oblasti automatizovaného vytváření GUI.
2. Formulace požadavků na automatizované a poloautomatizované generování GUI.

3. Analýza různých možných přístupů ke generování GUI pro funkcionální jazyky.

4. Návrh konceptu automatizovaných transformací anotovaného zdrojového kódu do modelu grafického uživatelského rozhraní.

5. Výběr vhodné implementační platformy (programovací jazyk, framework).

6. Implementace modelu grafického rozhraní.

7. Implementace grafické části uživatelského rozhraní v některém z běžných GUI frameworků.

8. Formulace a implementace případové studie.

9. Formulace závěrů.

Část II

Rešerše

Kapitola 3

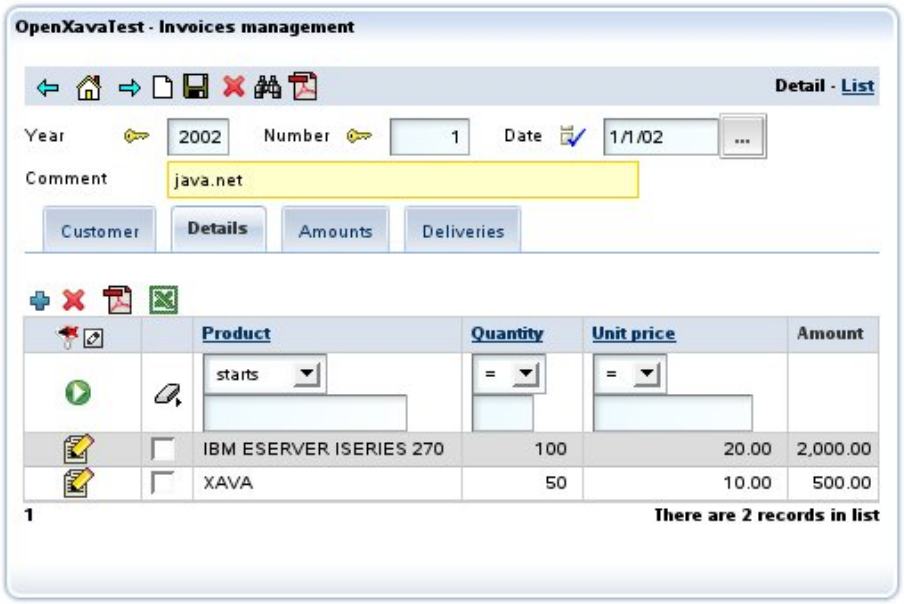
Současné metody generování GUI

Tato kapitola analyzuje současný stav na poli automatického generování grafického uživatelského rozhraní. Autorovi této práce není dosud známa žádná implementace takového generátoru pro funkcionální jazyky. Z praktických implementací lze tedy srovnávat pouze ty, které jsou určené pro jazyky objektově orientované. Je však také možné se inspirovat prací [9], jež implementuje generátor GUI pro jednoduchý výpočetní model přepisování výrazů.

Větších či menších systémů pro automatické generování uživatelského rozhraní pro OO jazyky existuje mnoho, většina z nich však sdílí podobné paradigma. Jejich fungování je založeno na anotaci doménového datového modelu pomocí nějakého systému metadat. Takto anotovaný model pak slouží jako vstup pro generátor, který vygeneruje nejen uživatelské rozhraní, ale mnohdy i aplikační logiku, databázové schéma a vůbec celou aplikaci. Někdy naopak ani statický generátor není potřeba a GUI pracuje s anotovaným modelem dynamicky například použitím reflexe.

3.1 OpenXava

Hlavním reprezentantem výše uvedeného přístupu ke generování GUI a business aplikací vůbec, může být framework OpenXava [10]. Tento framework je určen pro generování webových aplikací v jazyce Java s uživatelským rozhraním založeným na technologii AJAX. Vstupem pro generátor je doménový datový model napsaný



Obrázek 3.1: Příklad grafického rozhraní vygenerovaného pomocí generátoru OpenXava (převzato z [10]).

v jazyce Java a anotovaný pomocí kombinace anotací z JPA (Java Persistence Framework) a samotného OpanXava systému. Výstupem pak je webová aplikace, která pro ukládání dat používá Hibernate nebo právě JPA. Příklad anotované třídy, která představuje fakturu v nějakém informačním systému, ukazuje výpis 3.1. Z takto anotovaného modelu pak OpenXava generátor vygeneruje rozhraní, které ukazuje obrázek 3.1.

OpenXava je bezesporu velmi užitečnou technologií, která umožňuje vyvíjet business logiku aplikace zcela odděleně od uživatelského rozhraní. Její výhody a nevýhody jsou vesměs společné všem zde zmiňovaným objektově orientovaným frameworkům. Mezi hlavní výhody patří:

1. Nezávislost na konkrétním rozhraní. Jelikož se uživatelské rozhraní nevytváří v závislosti na konkrétní technologii, je možné implementovat generátory, které produkují UI využívající různých grafických knihoven na různých platformách od webových aplikací přes desktopové až k mobilním.

2. Produktivita. Vývojáři jsou osvobozeni od časově náročných úprav grafického rozhraní případně od vytváření různých variant pro různé platformy.

Výše popsaná technologie však přináší také nevýhody

Výpis 3.1 Příklad anotací modelu ve frameworku OpenXava (převzato z [10]).

```
@Entity
@View(
  members = "year, number, date;" +
    "comment;" +
    "customer { customer }" +
    "details { details }" +
    "amounts { amountsSum; vatPercentage; vat }" +
    "deliveries { deliveries }"
)
public class Invoice {
  @Column(length=4) private int year;
  @Column(length=6) private int number;
  private Date date;
  @Column(length=80) private String comment;
  @ManyToOne private Customer customer;

  @OneToMany(mappedBy="invoice")
  @ListProperties(
    "serviceType, product.description," +
    "product.unitPriceInPesetas, quantity," +
    "unitPrice, amount")
  private Collection<InvoiceDetail> details;

  @OneToMany(mappedBy="invoice")
  private Collection<Delivery> deliveries;
  // Getters and setters
  ...
  // Calculated properties
  @Digits(integerDigits=12, fractionalDigits=2)
  public BigDecimal getAmountsSum() { ... }
  @Digits(integerDigits=12, fractionalDigits=2)
  public BigDecimal getVat() { ... }
  @Digits(integerDigits=12, fractionalDigits=2)
  public BigDecimal getTotal() { ... }
}
```

1. Zdrojový kód doménového modelu je často zahlcen metadaty, která znesnadňují orientaci v něm a případné úpravy.

2. Uživatelské rozhraní by spíše mělo sledovat logiku chování uživatele a přizpůsobit se jeho intuici a pracovním návykům spíše než kopírovat business logiku aplikace [32].

3.2 Další OO systémy

Framework OpenXava byl použit jako zástupce mnoha dalších, které fungují na podobném principu. Zde jsou uvedeny jen některé z nich spolu s krátkými charakteristikami a případnými odlišnostmi.

NakedObjects [11] je framework pro platformu .NET, který opět pracuje s doménovým modelem tentokrát však napsaným v jazycích, které jsou určeny pro Common Language Runtime. NakedObjects narozdíl od frameworku OpenXava místo anotací využívá reflexi a uživatelské rozhraní vytváří plně dynamicky.

RomaFramework [12] pracuje s doménovým modelem v jazyce Java. Umožňuje však kromě anotací přímo v kódu používat také separátní XML soubory. Oba tyto přístupy je možné používat současně, což umožňuje zásadně zredukovat množství anotací, které znepřehledňují samotný kód.

Magritte [13] je zaměřen na dynamické webové aplikace v jazyce Smalltalk. Doménové třídy se anotují pomocí takzvaných popisných objektů, které pomocí jmenných konvencí přidávají k třídám dodatečné informace. Ty pak určují chování grafického rozhraní, specifikují databázová schémata apod.

Mezi další OO systémy pracující na podobných principech, jako jsou výše zmíněné, patří Trails [14], JMatter [15], Apache Isis [16] a další. Založení všech těchto i výše popsaných frameworků se však hodně blíží principu Model Driven Architecture, kde je celá aplikace včetně grafického rozhraní vygenerována z nějakého modelu. Jak však plyne z výše provedeného rozboru, pro takovýto přístup je často potřeba velmi expresivní systém anotací. Ten pak tvoří samostatný jazyk, jenž popisuje, jakým způsobem se mají prvky modelu zobrazovat, specifikuje

různé omezující podmínky a podobně. Tato práce se však ubírá jiným směrem, jenž je založen spíše na analýze zdrojového kódu, než na transformaci modelů. Tuto myšlenku rozvíjí práce [9], které se blíže věnuje následující část.

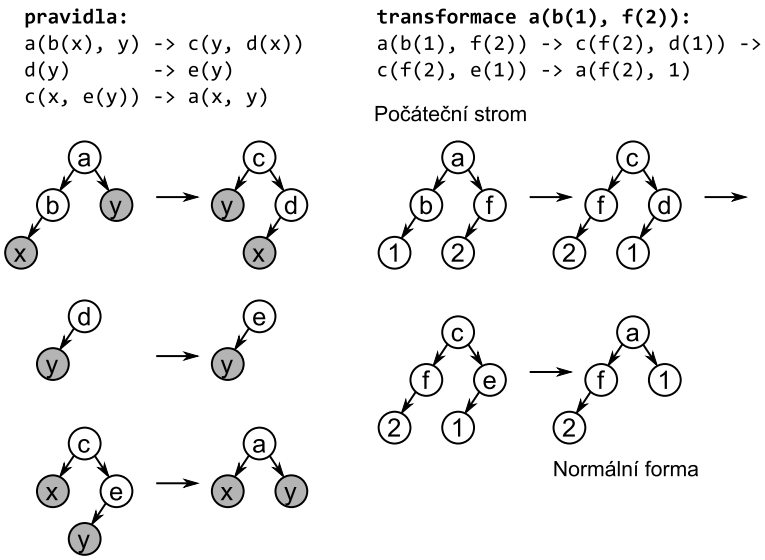
3.3 Programy založené na přepisování výrazů

Předešlé systémy pro generování GUI pocházely z objektově orientovaného programování a využívaly především anotovaného doménového modelu. Zajímavým počinem je však také práce [9], která se zabývá generováním GUI pro programy popsané v jednodušším výpočetním modelu. V této práci je program reprezentován pomocí přepisovacího systému, konkrétně založeného na přepisování stromů (též přepisování výrazů) [17]. Jelikož jde o daleko jednodušší reprezentaci programu než pomocí OOP, mohou tyto myšlenky posloužit jako zajímavé východisko pro tuto diplomovou práci.

3.3.1 Přepisování výrazů a generování GUI

Přepisování výrazů je jednoduchý výpočetní model založený na přepisování podvýrazů nějakého výrazu podle určitých přepisovacích pravidel. Tento model se zřídka používá jako základ pro nějaký programovací jazyk, má však mnoho aplikací v teoretické informatice [17]. Co se týče praktických využití, je možné zmínit například jazyk XSLT, jenž je také založen na jisté formě přepisování výrazů [18].

Na výrazy, s nimiž tento výpočetní model pracuje, je možné nahlížet jako na libovolně rozvětvené stromy. Výpočet pak probíhá transformací počátečního stromu postupnou aplikací přepisovacích pravidel. Pokaždé, když v aktuálním stromu existuje nějaký podstrom, který vyhovuje levé straně nějakého přepisovacího pravidla, tento podstrom se nahradí pravou stranou vybraného pravidla. Takto se pokračuje až do chvíle, kdy už není možné žádné pravidlo aplikovat. Výsledný strom je pak v takzvané normální formě. Příklad takového výpočtu ilustruje obrázek 3.2. Uzly s šedým pozadím zde reprezentují proměnné, kterým lze přiřadit libovolný podstrom.



Obrázek 3.2: Příklad přepisování výrazů (s úpravami převzato z [9]).

V práci [9] je prezentována myšlenka obohatit zpracovávané výrazy o anotace, které by napovídaly strukturu grafického uživatelského rozhraní. Lze tak označit místo, kde je od uživatele například vyžadováno celé číslo nebo kliknutí na tlačítko. Vzhledem k jednoduchosti tohoto modelu pak může být automatickým analyzátorem odvozena struktura podobná vývojovému diagramu, která popisuje možné cesty, jimiž se běh programu ubírá v závislosti na vstupu od uživatele. Z této struktury je pak možné automaticky vygenerovat abstraktní popis GUI, který specifikuje množinu dialogů a jejich ovládacích prvků pro zadaný program. Tento popis lze pak přímočaře převést na implementaci v nějaké konkrétní grafické knihovně.

Výše popsaný přístup se pro praktický vývoj aplikací nehodí už proto, že přepisování výrazů je příliš jednoduchý model pro vývoj větších a komerčních programů. Autoři práce [9] však předesílají, že na jimi představených principech by mohl být založen automatický generátor GUI i pro běžnější a v praxi používanější jazyky. Tato idea jistě stojí za prozkoumání a v následující kapitole bude rozebrána do větších podrobností především v souvislosti s funkcionálními jazyky jako Lisp či Clojure.

Kapitola 4

Generování GUI pro funkcionální jazyky

V dnešní době je velmi rozšířeným programovacím paradigmatem objektově orientované programování. Použití objektů jako základních stavebních kamenů programu přináší nesporně řadu výhod, užitečných technik a principů. Mezi nejčastěji jmenované patří zapouzdření, dědičnost či polymorfismus [22]. Přesto se stále častěji ozývají kritické hlasy směrem k OO programování jak z řad akademiků tak z řad softwarových inženýrů [23], [24], [25]. Do popředí se zároveň dostávají myšlenky známé z funkcionálních jazyků, jako je Lisp či Haskell. V důsledku toho různé OO jazyky jako je C++, C# nebo Java pomalu asimilují některé funkcionální principy a nadstavby. Do praxe se však začínají dostávat i nové funkcionální jazyky, jejichž ambice míří tak vysoko, jako je vývoj celých enterprise aplikací napsaných funkcionálně namísto objektově. Jedním z takových je například jazyk Clojure, o kterém blíže pojednává kapitola 5.

Následující část této práce popisuje, v čem spočívají výhody funkcionálních jazyků. Tyto výhody zároveň představují část motivace, proč se tato práce věnuje generování GUI právě pro funkcionální struktury. Zároveň tak na povrch vyplynou rozdílnosti v řešení tohoto problému pro objektově orientované a funkcionální jazyky.

4.1 Úskalí objektově orientovaného myšlení

Lze si představovat, že reálný svět se skládá z různých více či méně přesně vymezených konkrétních či abstraktních "objektů", jimž lze přisuzovat jisté vlastnosti a definovat operace, které s těmito objekty pracují. Z tohoto pohledu lze objektově orientované programování považovat za velmi užitečný způsob, jak modelovat realitu. Tento pohled však s sebou nese i značné problémy a nevýhody. Mnohé z nich jsou hlouběji rozebírány v [23] a [24]. Zde jsou diskutovány především ty, které nějak ovlivňují generování GUI a automatickou analýzu zdrojového kódu.

4.1.1 Mutabilita

Jednou z hlavních odlišností funkcionálního programování od OOP je takzvaná imutabilita. Její protiklad mutabilita nebo-li měnitelnost znamená, že atributy objektů v paměti je možné za života objektu měnit. Lze například přidávat prvky do objektu "seznam", který pořád zůstává tím samým objektem, i když se počet jeho prvků mění. Mutabilita však vede k překrývání pojmů stav a identita, což vede k zásadním problémům. Jedním z nich je často velká závislost běhu programu na aktuálním stavu mnoha různých objektů, které se navzájem ovlivňují a mění. Takto vzniká takzvaný špagetový kód, ve kterém je velmi těžké se orientovat a kdy je složité zjistit, co bude v konkrétní okamžik daná metoda dělat, pokud bude zavolána s určitými parametry. To samozřejmě velmi stěžuje jakoukoliv automatickou analýzu zdrojového kódu zaměřenou na chování výsledné zkompilované aplikace. Tento problém je možné zčásti odstranit disciplinovaným dodržováním určitých doporučených postupů pro psaní objektových aplikací. Funkcionální programování však tento problém odstraňuje už samotnou svojí podstatou.

4.1.2 Složitá syntaxe

Co se týče komplikovanosti syntaxe, zde se nabízí především srovnání s jazykem Lisp. Zdrojový kód napsaný v Lispu lze bezesporu parsovat pomocí daleko jednodušší gramatiky než například jazyk Java. Díky homoikonicitě[1] jazyka Lisp je

[1]Homoikonicita je vlastnost, kdy je program reprezentován přímo pomocí svých vlastních primitivních datových struktur.

možné k analýze zdrojového kódu využít přímo jeho základních prostředků pro práci se seznamy. Naopak v případě imperativních a OOP jazyků je taková analýza značně náročnější. Přestože na začátku měly být právě OOP jazyky ty, které budou jednoduché a bude snadné se je naučit, postupem času se kvůli neustálému přidávání nových vlastnosti od jednoduchsti velmi vzdálily [25].

4.2 Funkcionální paradigma a jazyk Clojure

Vzhledem k výše popsaným důvodům a ve světle analýzy provedené v kapitole 3 je vhodné poohlédnout se po jednodušším jazyku, který by ovšem zároveň byl prakticky použitelným a používaným i k vývoji komerčních a business aplikací. Například výpočetní model založený na přepisování stromů zmíněný v předchozí kapitole těmto požadavkům nevyhovuje, neboť je již příliš jednoduchý. Zajímavým kompromisem tak může být použití nějakého jazyka funkcionálního. Jejich syntaxe je jednoduchá a používají neměnitelné datové struktury, což obojí zjednodušuje analýzu kódu. Zároveň však mezi nimi lze najít takové, které se používají přímo v praxi i k vývoji větších aplikací. Jedním z takových jazyků je poměrně nový funkcionální jazyk vycházející z Lispu jménem Clojure. Tento jazyk je vyvíjen s maximálním důrazem na použitelnost v praxi, tedy především pro psaní rozsáhlejších aplikací mimo obory matematiky či umělé inteligence.

Kapitola 5

Jazyk Clojure

Clojure je dynamický funkcionální programovací jazyk běžící nad virtuálním strojem jazyka Java (Java Virtual Machine – JVM). Je vytvořen s cílem být obecně použitelným jazykem kombinujícím výhody interaktivního skriptovacího jazyka s efektivní a robustní infrastrukturou pro vícevláknové programování. Clojure se kompiluje přímo do JVM byte kódu, přesto však zůstává plně dynamický. Všechny schopnosti a možnosti podporované tímto jazykem jsou přístupné a použitelné za běhu vytvářeného programu. Touto vlastností je Clojure podobný například Smalltalku z objektově orientovaného světa.

Clojure je dialektem Lispu. S tímto jazykem sdílí filozofii homoikonicity, kdy kód a data jsou jedno a totéž, a taktéž mocný systém pro psaní maker. Clojure navazuje na myšlenky typické pro funkcionální jazyky, jako jsou neměnitelné a perzistentí datové struktury a jasné odlišení stavu a identity. Mutabilita a správa stavu jsou však přes všechny své nevýhody mnohdy užitečné i ve funkcionálním světě, a proto jazyk Clojure implementuje takzvanou softwarovou transakční paměť (software transactional memory – STM) [31]. Ta umožňuje každou změnu stavu zabalit do atomické transakce. Tímto způsobem jsou všechny změny stavu v aplikaci kontrolované a jasně izolované od zbytku funkcionálního kódu, což jednak zkvalitňuje strukturu aplikace a jednak umožňuje daleko snadněji využívat paralelismu.

Zásadní výhodou jazyka Clojure je jeho schopnost těsně spolupracovat s jazykem Java. Jelikož programy v Clojure běží na virtuálním stroji Javy, je možné

z nich volat všechny knihovny jazyka Java, ale také rovnou vytvářet Javovské třídy, implementovat rozhraní, či používat Javovské primitivní typy. Takto je možné snadno využívat veškerou funkcionalitu, kterou Javovské knihovny a frameworky nabízejí, například souborový vstup a výstup, vícevláknové programování, přístup k databázím, webové aplikace a v neposlední řadě tvorbu GUI. Díky této těsné interoperabilitě je Clojure použitelnější pro psaní moderních aplikací než jiné dialekty Lispu.

Následuje popis jazyka Clojure, jehož cílem je seznámit čtenáře s jeho základními myšlenkami, syntaxí a především odlišnostmi od jiných běžných variant Lispu, jako jsou Common Lisp nebo Scheme. Tato kapitola je psána s přepokladem, že čtenář této práce má alespoň základní povědomí o funkcionálním programování a jazyku Lisp, případně zkušenosti s jeho dialekty Common Lisp či Scheme. Takové znalosti lze načerpat například z dnes již klasické knihy [19] a knih [20], [21]. Pro hlubší studium jazyka Clojure lze doporučit publikace [28], [29] nebo [30]. V době psaní této práce bohužel neexistuje žádná oficiální specifikace tohoto relativně nového jazyka, na kterou by bylo možné se v textu odkazovat. Podrobný popis všech základních myšlenek lze nalézt na oficiálním webu [26]. Dokumentaci všech dostupným funkcí v jazyce lze pak najít na [27].

5.1 Clojure jako dialekt Lispu

Ve srovnání s dialekty Common Lisp a Scheme je Clojure blíže druhému v pořadí. Clojure je takzvaný LISP-1, tedy funkce a hodnoty sdílejí tentýž jmenný prostor. Na druhou stranu se však od jazyka Scheme také v mnohém odlišuje. Množství odlišností přímo vyplývá s jeho těsné vazby na Javu.

Základní principy mají Scheme a Clojure společné. Kód se zapisuje pomocí seznamů v kulatých závorkách v prefixové notaci (takzvané S-výrazy). Každá “operace” je implementována buď jako funkce, makro nebo takzvaná speciální forma. Téměř všechny vestavěné funkce a makra jsou implementovány přímo v jazyce Clojure. Speciální formy jsou naopak přímo zpracovávány kompilátorem. Takovýchto forem je poměrně málo a další nelze přidávat. Přehled základních speciálních forem jazyka Clojure a jejich ekvivalentů v jazyce Scheme ukazuje tabulka 5.1.

Jméno formy	**Popis**	**Ekvivalent ve Scheme**
`fn`	Definice anonymní funkce.	`lambda`
`def`	Vazba symbolu v aktuálním prostředí.	`define`
`let`	Vazby symbolů v novém prostředí.	`let`
`if`	Jednoduché větvení programu.	`if`
`do`	Sekvenční vyhodnocování.	`begin`
`loop, recur`	Rekurzivní volání, které nezvětšuje zásobník[1].	není
`quote`	Zabraňuje vyhodnocení výrazu.	`quote`
`.` (tečka)	Volání metod z Javy.	není
`new`	Volání konstruktoru z Javy.	není
`throw, try, catch, finally`	Práce s výjimkami.	není

Tabulka 5.1: Srovnání speciálních forem jazyka Clojure s jazykem Scheme.

Ostatní speciální formy nejsou v této práci využívány nebo jsou určeny pouze pro interní účely jazyka Clojure.

Clojure nabízí literály pro základní primitivní datové typy, jako jsou celá a desetinná čísla s libovolnou velikostí a přesností. Nechybí také literály pro zlomky, boolovské hodnoty, znaky a řetězce. Tyto typy jsou kromě zlomků přímo třídami jazyka Java jako `BigInteger`, `BigDecimal`, `Boolean`, `Character` a `String`. Samozřejmostí je také typ `symbol` typický pro jazyk Lisp. Co se však v jazyku Scheme nenachází jsou takzvaná *klíčová slova* (*keywords*).[2]. Ta se narozdíl od symbolů zapisují s dvojtečkou na začátku a jejich vyhodnocení vrátí opět totéž klíčové slovo. Mají vysoce optimalizovaný test na rovnost a slouží proto často jako klíče v mapách (o mapách viz dále). Také často velmi zpřehledňují zdrojový kód, kdy například

```
(circle :x 4.5 :y 7 :radius 2)
```

je jistě přehlednější než pouhé

```
(circle 4.5 7 2)
```

Speciální postavení má v Clojure hodnota `nil`, která ve skutečnosti znamená absenci jakékoliv hodnoty a je ekvivalentní hodnotě `null` z jazyka Java. Nejde tedy ani o symbol ani o alias pro prázdný seznam jako v Common Lispu. Každá kolekce v Clojure totiž má svůj vlastní symbol pro definování její prázdné instance. Píše se pak `'()` pro prázdný seznam, `[]` pro prázdný vektor, {} pro prázdnou mapu a podobně.

5.1.1 Kolekce

Čím se Clojure zásadně liší od Scheme jsou takzvané kolekce. Clojure kromě seznamů nativně implementuje další sekvenční struktury – vektory, mapy a množiny. Navíc seznamy jsou implementovány jako klasické spojené seznamy, nikoliv

[1] Virtuální stroj jazyka Java bohužel nepodporuje tail-call optimisation, takže Clojure potřebuje pro tento koncept explicitní konstrukci. Formy `loop` a `recur` nejsou pro tuto diplomovou práci významné, neboť všechny zdrojové kódy zde použité jsou napsány na vyšší úrovni abstrakce a tyto formy vůbec nepoužívají.

[2] Zde je termín 'klíčové slovo' použít v jiném významu než jak bývá běžně používán při popisu programovacích jazyků.

jako cons buňky známé z jiným dialektů Lispu. Se všemi těmito strukturami (a s mnohými dalšími) je možné jednotně pracovat pomocí abstrakce zvané *sekvence*. Clojure opouští klasické, avšak ne zcela srozumitelné funkce jako `car` a `cdr`, místo kterých zavádí `first` a `next`. Pokud nějaká struktura implementuje tyto dvě funkce, je to takzvaná sekvence. Tyto dvě funkce jsou tedy velmi obecné a díky nim je možné s mnoha sekvenčními datovým strukturami pracovat jednotným způsobem.

5.1.2 Vektory

Nejprve je třeba zmínit, že vektor v Clojure představuje jinou strukturu než stejně pojmenovaný pojem v jazyce Scheme. Předně délka vektoru není fixní, čili lze s vektory pracovat podobně jako se seznamy. Chovají se tedy spíše podobně jako natahovací pole. Vzhledem k nutnosti persistence není přístup k n-tému prvku konstantní, nýbrž trvá $log_{32}N$ kroků, kde N je délka vektoru. Vektory se zapisují do hranatých závorek, například

```
[0 1 2 3 4 5 6 7 8 9]
```

Prvky k vektoru se narozdíl od seznamů přirozeně přidávají na konec a navíc lze jejich velikost zjistit v konstantním čase.

Vektory slouží ke dvěma hlavním účelům. Za prvé zpřehledňují zápis kódu, neboť díky hranatým závorkám opticky narušují záplavu kulatých závorek typickou pro Lisp. Například zápis lambda funkce počítající aritmetický průměr pak vypadá takto

```
(fn [x y] (/ (+ x y) 2))
```

kde jsou díky vektoru jasně odděleny parametry funkce od jejího těla. Za druhé, vektory jsou díky odlišné časové složitosti operací nad nimi užitečné tam, kde jsou operace se seznamy příliš pomalé.

5.1.3 Mapy

Mapa je další struktura, která jednak velmi ulehčuje psaní kódu, jednak umožňuje vytvářet různorodější a efektivnější datové struktury. Mapy efektivně zobrazují klíče na hodnoty. Zapisují se do složených závorek, kde vždy po dvojicích po sobě následují klíče a hodnoty. Například mapa

```
{:top-left [12.5 14.3] :width 5 :height 10}
```

velmi srozumitelně popisuje obdélník. Mapy jsou v Clojure často implementovány jako hashovací tabulky a jako klíčů se často používá klíčových slov, jak to ukazuje příklad s obdélníkem. Lze s nimi však také pracovat jako se sekvencemi dvojic [klíč, hodnota]. Zajímavá je možnost použít klíčové slovo na místě volání funkce k získání hodnoty příslušné k danému klíči v mapě, například

```
(:width rectangle)
```

vrátí číslo 5, odkazuje-li symbol `rectangle` na výše uvedenou ukázkovou mapu.

5.1.4 Množiny

Poslední datovou strukturou přímo zabudovanou do syntaxe jazyka Clojure jsou množiny. Jak název napovídá, množina nemůže obsahovat dva stejné prvky. Množiny se zapisují do kulatých závorek s prefixem #. Tedy například

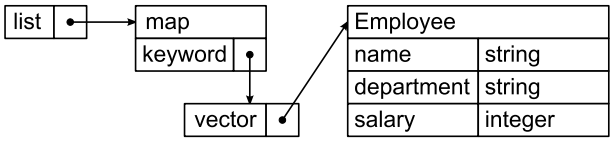
```
#{"Pavel" "Martin" "Radek" "Michal"}
```

Clojure samozřejmě poskytuje různé funkce pro práci s množinami jako sjednocení, průnik apod.

5.1.5 Datové typy

Datové typy v jazyce Clojure se chovají podobně jako mapy. Mají však pevně daný počet a pořadí položek a jejich klíče jsou klíčová slova. Lze tedy definovat například datový typ pro zaměstnance

```
(defrecord Employee [name department salary])
```



Obrázek 5.1: Příklad strukturového diagramu.

Instance datového typu se vytváří zavoláním jeho jména s tečkou na konci a následným seznamem hodnot jeho atributů. Potom lze s takovým typem pracovat téměř stejně jako s mapou. Například takto lze vytvořit zaměstnance a zjistit jeho plat:

```
(def jan (Employee. "Jan Novák" "Human resources" 25000))
(:salary jan)
```

Datové typy jsou důležité především ve spojení s takzvanými *protokoly*, které ovšem v této práci používány nejsou. Zde jsou tedy datové typy použity hlavně pro zpřehlednění některých datových struktur a funkcí.

5.1.6 Strukturové diagramy

V této práci bude často potřeba popsat, jak vypadá nějaká komplikovanější složená datová struktura. Slovní popisy jako například "seznam map, které mapují klíčová slova na vektory zaměstnanců" jsou neohrabané, těžko čitelné a brzy přestanou stačit. Ke znázornění takvých struktur proto bude často použito takzvaných *strukturových diagramů*, které znázorňují složitější struktury daleko přehledněji. Obrázek 5.1 ukazuje strukturový diagram pro výše zmíněnou slovně popsanou strukturu.

5.2 Základy programování

V následující části jsou na několika příkladech vysvětleny základní principy psaní kódu v jazyce Clojure. Jak už bylo řečeno, Clojure má mnoho společného s jazykem Scheme, proto čtenáři znalému tohoto dialektu bude mnohdy stačit pouze přivyknout novým jménům některých základních funkcí.

Výpis 5.1 ukazuje známý třídící algoritmus QuickSort implementovaný v Clojure. Tato ukázka je uvedena jako první, neboť na malém prostoru ukazuje většinu základních konstrukcí jazyka. Zároveň je možné si plně vychutnat eleganci funkcionálního programování.

Výpis 5.1 Třídící algoritmus QuickSort implementovaný v Clojure.

```
(defn quick-sort [[m & coll]]
  (cond (nil? m)    []
        (nil? coll) [m]
        :else       (concat
                      (quick-sort (filter #(<= % m) coll))
                      [m]
                      (quick-sort (filter #(> % m) coll)))))
```

5.2.1 Funkce

Nové funkce se vytváří pomocí makra `defn`. Parametry funkce se zapisují jako vektor do hranatých závorek. Při specifikaci parametrů je možné navíc použít takzvaný *destructuring*. Každý parametr tak může být zapsaný jako vektor (očekává-li se na jeho místě sekvence) nebo mapa (očekává-li se asociativní struktura). Takto je například možné rozložit sekvenci ve vstupním parametru `coll` na první prvek, druhý prvek a zbytek

```
[first second & rest :as coll]
```

Díky klíčovému slovu `:as` tak ani nedojde ke ztrátě jména vstupní sekvence jako celku.

Lambda funkce se plným zápisem vytvářejí pomocí formy `fn`, tedy například

```
(fn [x y] (/ (+ x y) 2))
```

je lambda funkce počítající aritmetický průměr. Jelikož lambda funkce se používají velmi často, je možné je zkráceně zapisovat také takto:

```
#(/ (+ %1 %2) 2)
```

Symboly začínající znakem procenta označují postupně jednotlivé parametry. Pokud má funkce pouze jeden parametr, stačí psát pouze `%`. Tohoto zápisu je využito v implementaci QuickSortu k filtrování prvků podle velikosti.

Výpis 5.2 ukazuje implementaci dalšího známého třídícího algoritmu, tentokrát MergeSortu. Zde je dobré si všimnout, jak lze destructuring používat i v makru `let`.

Výpis 5.2 Třídící algoritmus MergeSort implementovaný v Clojure.

```
(defn -merge [left right]
  (cond (nil? left)  right
        (nil? right) left
        :else        (let [[x & rleft] left
                           [y & rright] right]
                       (if (<= x y)
                         (cons x (-merge rleft right))
                         (cons y (-merge left rright))))))

(defn merge-sort [[x & rcoll :as coll]]
  (cond (nil? x)     []
        (nil? rcoll) [x]
        :else        (let [[left right] (split-at
                                         (/ (count coll) 2)
                                         coll)]
                       (-merge (merge-sort left)
                               (merge-sort right)))))
```

5.3 Synchronizační primitiva

Jedním z hlavních stavebních bloků funkcionálně orientovaných programů jsou takzvaně čisté funkce, tedy takové, které nemají žádné vedlejší efekty ani jejich výsledek nezávisí na stavu programu. Ne všechny programy však mohou být jedna velká funkce s jasně definovaným vstupem a výstupem, jako jsou například kompilátory, automatické dokazovače vět nebo různé UNIXové utility. Alespoň

některé části většiny programů potřebují nějak zacházet se stavem. Takovéto části by však měly být co nejmenší a co nejjasněji odděleny od čistě funkcionálních částí programu. Jazyk Clojure za tímto účelem implementuje takzvanou softwarovou transakční paměť [31] a zavádí tři synchronizační primitiva – reference, atomy a agenty – jejichž úkolem je dát práci se stavem jasné hranice a metodiku. Tyto primitiva obsahují hodnoty, které je možné měnit a pomocí transakcí také atomicky synchronizovat. To má výhodu nejen v jasném oddělení práce se stavem, ale hlavně je tak možné psát v Clojure programy intenzivně využívající paralelizaci a práci v mnoha vláknech bez nutnosti používat složité a často nepřehledné uzamykací mechanizmy.

Jelikož programy v této práci si vystačí se synchronizací stavu v rámci jednoho vlákna, synchronizační primitiva zde budou vysvětlena jen stručně. Zájemci o tuto problematiku mohou využít knih [28] a [29], kde jsou transakční mechanizmy a primitiva vysvětleny velmi podrobně.

5.3.1 Atomy, reference a agenti

Každé ze tří výše zmíněných primitiv má specifické vlastnosti a každé je určeno k použití v jiné situaci. Jednou z takových situací je nutnost atomické koordinované změny dvou na sobě závislých stavů. Například při transferu peněz z jednoho účtu v bance na jiný je třeba najednou provést odečtení peněz z prvního účtu a přičtení na druhý účet. Koordinovanou změnu stavů umožňují reference. Pokud není třeba stavy koordinovat – jsou takzvaně nezávislé – je lepší použít agenty nebo atomy, neboť změny nezávislých stavů jsou často rychlejší [28].

Další situace, která je běžná při manipulaci se stavem, je nutnost asynchroních změn. Asynchroní změna stavu nastává, pokud funkce, jež daný stav mění, vrátí řízení zpět bez čekání, až se změna skutečně provede. Provedení změny může nastat v nějakém okamžiku v budoucnu v jiném vlákně, zatímco hlavní vlákno pokračuje v běhu. Asynchroních změn stavů se široce využívá při paralelních výpočtech nebo například v propracovaných systémech založených na posílání událostí [28]. Agenti jsou v jazyce Clojure primitivum, které řeší právě tuto situaci. V této práci však agentů nebude třeba.

Reference

Reference slouží ke koordinovaným změnám více stavů pomocí transakcí. Transakce je posloupnost operací, která se chová jako jedna atomická operace. Transakce v Clojure jsou neblokující a optimistické. To znamená, že jedna transakce nikdy nečeká až skončí jiná, která náhodou mění stejný stav. Pokud takový případ nastane, první transakce přečte nový obraz stavů, se kterými pracuje, a restartuje se. Restartování transakcí umožňují kromě referencí také atomy.

Reference se vytvářejí použitím funkce `ref`. Takto lze například vytvořit referenci obsahující hodnotu 10.

```
(def my-ref (ref 10))
```

Jazyk Clojure má speciální syntax pro přístup k hodnotám, na které jednotlivá primitiva odkazují, založenou na symbolu `@`. Hodnotu výše vytvořené reference tak lze získat zpět zápisem `@my-ref`. Ke změnám hodnot referencí se používají funkce `ref-set` a `alter` [27]. Funkce `ref-set` nastavuje referenci novou hodnotu nezávislou na hodnotě předchozí. Výše vytvořenou referenci tak lze nastavit na hodnotu 20 takto:

```
(ref-set my-ref 20)
```

Pokud je třeba vzít v úvahu předchozí hodnotu, použije se funkce `alter`. Ta jako svůj parametr očekává funkci, která nějak transformuje původní hodnotu reference. Takto se například hodnota reference změní na její dvojnásobek

```
(alter my-ref #(* % 2))
```

Funkce pro změnu referencí je však možné spouštět pouze uvnitř transakcí. Spuštění transakce se pak provádí pomocí makra `dosync` [27]. Toto makro očekává posloupnost funkcí, které mají dohromady tvořit jednu transakci.

Atomy

Atomy jsou nejjednodušší synchronizační primitivum. Slouží k nekoordinovaným synchroním změnám nějaké hodnoty a není třeba je proto spouštět v transakcích. Atomy se vytvářejí podobně jako reference, jen místo symbolu `ref` se použije

	Reference	Agenti	Atomy
Koordinované	✔		
Asynchroní		✔	
Restartovací	✔		✔

Tabulka 5.2: Vlastnosti synchronizačních primitiv v Clojure. (Částečně převzato z [29].)

symbol `atom`. Jejich změny se také dějí podobně jako u referencí. Místo funkce `ref-set` se ale použije `reset!` a funkci `alter` nahrazuje funkce `swap!` [27].

Tím jsou možnosti synchronizačních primitiv v Clojure vyčerpány. Jejich vlastnosti popsané výše lze shrnout do přehledné tabulky 5.2. V implementační části této práce jsou použity atomy a reference k synchronizaci stavu v datovém modelu grafického rozhraní. Je tak potom možné lépe zkombinovat grafické rozhraní, které je ze svého principu velmi závislé na stavu, s funkcionální logikou aplikace, která by se měla změnám stavu naopak co nejvíce vyhýbat.

5.4 Multimetody a typové hierarchie

Zcela zásadní konstrukcí v jazyce Clojure jsou takzvané *multimetody*. Díky multimetodám mohou programy napsané v Clojure využívat takzvaného runtime polymorfismu v daleko obecnější podobě, než je polymorfismus založený na dědičnosti a virtuálních funkcích známý z objektově orientovaných jazyků, jako jsou Java nebo C++. Clojure totiž odděluje typové hierarchie od implementací metod, které definují chování a vlastnosti jednotlivých typů. Díky tomu je například možné vytvořit několik nezávislých typových hierarchií nad stejnou množinou typů. Tyto hierarchie mohou navíc bez problému využívat vícenásobné dědičnosti a tak lépe modelovat vztahy v reálném světě.

5.4.1 Multimetody

Multimetody slouží k tomu, aby bylo možno za běhu programu vybrat z více implementací pro tutéž funkci podle hodnot předávaných parametrů. Implemen-

tace, která se má pro konkrétní hodnoty parametrů zavolat, se vybere voláním takzvané *dispatch funkce*. Tato funkce je definována při deklaraci multimetody. Velmi ilustrativní příklad použití multimetod lze nalézt v knize [28]. Zde je uveden v poněkud zkrácené podobě.

Předpokládejme, že vyvíjeným programem je fantasy počítačová hra. V té se mohou vyskytovat například následující postavy.

```
(def a {:name "Arthur", :species ::human, :strength 8})
(def b {:name "Balfor", :species ::elf, :strength 7})
(def c {:name "Calis", :species ::elf, :strength 5})
(def d {:name "Drung", :species ::orc, :strength 6})
```

Pro jednotlivé rasy jsou použita takzvaná kvalifikovaná klíčová slova začínající dvojitou dvojtečkou. Klíčové slovo `::orc` je zkratkou pro

```
:<namespace>/orc,
```

kde `namespace` je jmenný prostor souboru, ve kterém se toto klíčové slovo vyskytuje. Kvalifikovaná klíčová slova jsou tedy vázána na konkrétní jmenný prostor a jsou tak vhodná k označování typů.

Úkolem multimetody `encouter` je implementovat akci, která nastane, jestliže se ve hře potkají dvě postavy.

```
(defmulti encounter (fn [x y]
                      [(:species x) (:species y)]))
(defmethod encounter [::elf ::orc] [elf orc]
  (str "Brave elf " (:name elf)
       " attacks evil orc " (:name orc)))
(defmethod encounter [::orc ::elf] [orc elf]
  (str "Evil orc " (:name orc)
       " attacks innocent elf " (:name elf)))
(defmethod encounter [::elf ::elf] [orc1 orc2]
  (str "Two elves, " (:name orc1)
       " and " (:name orc2)
       ", greet each other."))
```

Makro `defmulti` přiřazuje ke jménu nové multimetody její dispatch funkci. V tomto příkladu dispatch funkce vrací dvouprvkový vektor obsahující rasy postav, které se potkaly. Poté následují jednotlivé implementace pro konkrétní dvojice ras. Toto je příklad polymorfismu na základě typu obou parametrů funkce. Jelikož na dispatch funkci se nekladou žádná omezení, již z tohoto krátkého příkladu je zřejmé, jak širokou škálu možností multimetody v Clojure nabízejí.

5.4.2 Typové hierarchie

Narozdíl od objektově orientovaných jazyků jako Java nebo C++ nejsou typy v Clojure vázány na žádnou implementaci a typové hierarchie nejsou implicitně vytvořeny vztahem dědičnosti. K označení typů se často používá kvalifikovaných klíčových slov. Pomocí funkce `derive` lze pak explicitně vytvořit IS-A vztah mezi dvěma typy. Budeme-li pokračovat v započatém příkladu z knihy [28], můžeme například psát

```
(derive ::human ::good)
(derive ::elf ::good)
(derive ::orc ::evil)
(derive ::elf ::magical)
(derive ::orc ::magical)
```

Jak vidno, typ může být odvozen od více typů najednou. Multimetody umí s vytvořenými hierarchiemi pracovat a respektovat IS-A vztahy jimi definované. Je tedy možné přepsat metodu `encounter` tak, aby pracovala na základě typů `::good` a `::evil`.

```
(defmulti encounter (fn [x y]
                      [(:species x) (:species y)]))

(defmethod encounter [::good ::good] [x y]
  (str (:name x) " and " (:name y) " say hello."))

(defmethod encounter [::good ::evil] [x y]
  (str (:name x) " is attacked by " (:name y)))
```

```
(defmethod encounter [::evil ::good] [x y]
  (str (:name x) " attacks " (:name y)))

(defmethod encounter :default [x y]
  (str (:name x) " and " (:name y)
       " ignore one another."))
```

Multimetody jsou velmi mocným nástrojem jazyka Clojure a ve frameworku v implementační části této práce jsou hojně využívány například k velmi přímočaré a srozumitelné implementaci typového systému.

Část III

Vlastní řešení

Kapitola 6

Funkcionálně strukturované UI

Po rozboru současných možností v oblasti generování GUI, výběru Clojure jako vhodného implementačního jazyka a zevrubném popisu tohoto jazyka následuje kapitola, která představuje konkrétní přístup k automatickému generování GUI. Tato kapitola se zabývá myšlenkou funkcionálně strukturovaného uživatelského rozhraní (FSUI). Tento koncept vznikl po analýze různých možných přístupů ke generování uživatelského rozhraní z kódu funkcionálního programu. První část této kapitoly se věnuje této analýze. Na tu pak navazuje část druhá, která popisuje a vysvětluje samotný pojem FSUI.

6.1 Základní myšlenky

Hovoří-li se o automatickém generování grafického uživatelského rozhraní, je nutné nejdříve blíže vysvětlit, co se takovým generováním přesně myslí. Je třeba specifikovat jaký vstup by měl automatický generátor očekávat, jakým způsobem by měl pracovat a jaký výsledek by měl produkovat. Tyto myšlenky jsou rozvedeny v následujících odstavcích.

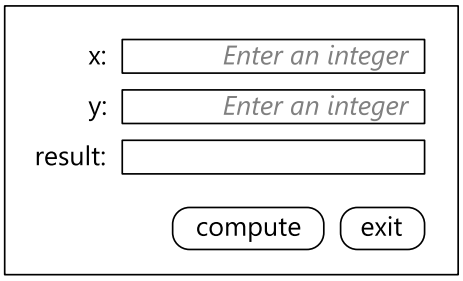
Hlavní idea automatického generování GUI vychází z předpokladu, že zdrojový kód programu, ať již funkcionálního či jiného, obsahuje množství informací, které z velké části určují způsob jeho ovládání a komunikace s uživatelem. Pokud se takový zdrojový kód na místech, která slouží k interakci s uživatelem, doplní o dodatečné informace specifické pro uživatelské rozhraní, lze pak takové

rozhraní z takto anotovaného kódu odvodit. Samozřejmě nelze tvrdit, že takové rozhraní bude pro koncového uživatele nějak výrazně komfortní. Cílem je však spíše vygenerovat takové rozhraní, pomocí kterého bude možné efektivně otestovat funkčnost programu, a které může případně sloužit jako prototyp pro zákazníka v počátečních fázích vývoje. Generátor produkující takové rozhraní může být plně automatický, pokud se nekladou zásadní požadavky na kvalitu výsledného UI například z hlediska zásad Human Computer Interaction (HCI) [32]. Sofistikovanější generátor pak může jednak nabídnout možnost ovlivnit výsledek pomocí dodatečných parametrů zadaných na vstupu, jednak může produkovat snadno spravovatelný a rozšiřitelný kód, jehož úpravami je pak možné grafické rozhraní vyladit na požadovanou kvalitu.

Grafické uživatelské rozhraní je způsob, jak ovládat počítačový program pomocí interaktivních grafických ovládacích prvků namísto zadávání textových příkazů do terminálu. V principu se však ovládání pomocí grafického rozhraní od komunikace s programem pomocí textového terminálu tolik neliší. V obou případech program čeká, až uživatel zadá nějaký vstup, který vyvolá programem definovanou odezvu. Je lhostejno, zda je takový vstup zadán jako textový příkaz nebo kliknutí na tlačítko. Počítačový program má v obou případech ve zdrojovém kódu definovány body, kde očekává vstup od uživatele. Tyto body mohou jasně specifikovat typ hodnoty nebo akce, která se od uživatele očekává. Například pseudokód části programu, který sčítá dvě celočíselné hodnoty, může vypadat takto:

```
Integer x, y, result;
result = InputInteger(x) + InputInteger(y);
OutputInteger(result);
```

Tento kód obsahuje dostatek informací k tomu, aby bylo možné automaticky vytvořit například dialog na obrázku 6.1. Tento příklad vede k představě generátoru, který by grafické rozhraní generoval na základě analýzy vstupních a výstupních bodů ve zdrojovém kódu. Jak už bylo zmíněno v kapitole 3, tomuto přístupu se velmi blíží práce [9]. Jelínek a Slavík zde sice používají velmi jednoduchý výpočetní model založený na přepisování výrazů, předesílají však, že by měl jít rozšířit i na komplikovanější modely, například právě funkcionální jazyky. Tato myšlenka rozhodně stojí za hlubší prozkoumání. Nazvěme takový přístup IO-orientovený.



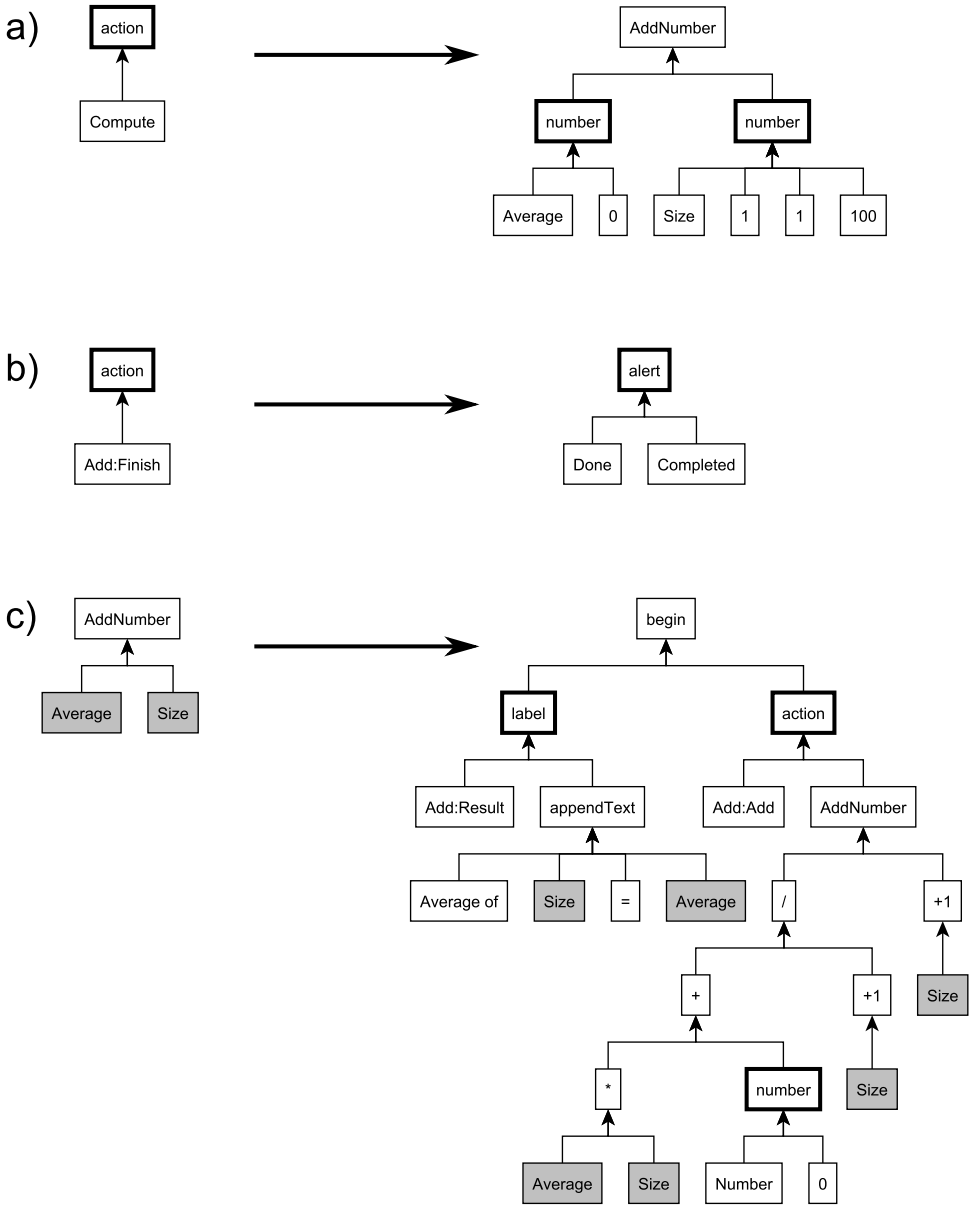
Obrázek 6.1: Příklad vygenerovaného GUI pro sčítání dvou celých čísel.

6.2 IO-orientovaný generátor

Prvotní představa fungování takzvaně IO-orientovaného generátoru spočívá v tom, že generátor statickou analýzou kódu vyhodnotí, jakými cestami se běh programu po spuštění ubírá a jaké vstupy na této cestě od uživatele vyžaduje. Zjistí-li pak generátor například to, že k výpočtu součtu je potřeba od uživatele získat dvě celá čísla, může tyto dva vstupy a jeden výstup sloučit do jednoho přehledného formuláře jako na obrázku 6.1. V práci [9] je uveden příklad programu, který postupně počítá aritmetický průměr z čísel zadávaných uživatelem. Tento příklad se i zde velmi hodí k demonstraci možností a úskalí takovéto techniky generování GUI.

Ukázkový program pracuje tak, že si nejdříve od uživatele vyžádá dvě čísla `average` a `size`, která představují počáteční průměr a počet čísel, ze kterých byl tento průměr vypočítán. Poté uživatel postupně zadává další čísla a program podle toho aktualizuje výsledný průměr. Přepisovací pravidla pro tento program jsou vyobrazena na obrázku 6.2. Je použito stejná konvence jako na obrázku 3.2, tedy šedé uzly představují proměnné. Uzly s bílým pozadím a slabým rámečkem jsou buď textové řetězce nebo čísla. Uzly se silným rámečkem jsou direktivy pro grafické rozhraní, které říkají, kdy je od uživatele vyžadováno číslo (`number`), kdy akce, tedy např. kliknutí na tlačítko (`action`) a kdy se mají uživateli zobrazit nějaké informace (`label`, `alert`).

Při bližším pohledu na strukturu těchto přepisovacích pravidel není těžké sledovat myšlenku programu a přepsat jej do Clojure (viz výpis 6.1). Díky syntaxi jazyka Lisp je kód organizován velmi podobně jako stromové struktury na obrázku 6.2. Lze tedy jistě pokračovat podobným směrem, jakým se vydává práce [9] a pokusit se vytvořit analyzátor, jenž by kód v jazyce Clojure převedl na strukturu,



Obrázek 6.2: Přepisovací pravidla pro ukázkový program počítající aritmetický průměr.

která popisuje běh programu a jeho větvení v závislosti na vstupech od uživatele. Tato práce se ovšem tímto směrem neubírá a to především z následujících důvodů, které ukazují nevýhody tohoto přístupu.

Výpis 6.1 Program pro výpočet průměru přepsaný do Clojure.

```
(defn add-number [average size]
  (do
    (label "Add:Result" (str "Average of " size " = " average))
    (action
      "Add:Add"
      (add-number
        (/ (+ (* average size) (number 0)) (inc size))
        (inc size))
      "Add:Finish"
      (alert "Done" "Completed"))))

(defn compute []
  (add-number (number 0)
              (number 1 1 100)))
```

1. Při pohledu na kód 6.1 je obtížné bez většího úsilí zjistit, co vlastně tento program dělá. Obsahuje totiž mnoho informací, které se vážou ke grafickému rozhraní. Tyto informace interferují se samotným kódem a zásadním způsobem snižují jeho přehlednost. To má také vliv na případné další úpravy tohoto kódu, neboť programátor tak musí upravovat nejen logiku aplikace ale také logiku uživatelského rozhraní.

2. Vinou těsné vazby na grafické rozhraní není možné tento program bez nějaké formy uživatelského rozhraní vůbec spustit, neboť přímo vnitřní části funkcí vyžadují interakci s uživatelem. Funkcionální programy je ovšem zvykem vyvíjet a testovat například z textového REPL rozhraní (Read-Eval-Print-Loop), které umožňuje přímo volat jednotlivé funkce na příkazovém řádku.

3. Čistě technickým problémem je pak přerušování běhu programu. Pokud se například zavolá funkce `add-number`, její průběh nemůže skončit, dokud uživatel nevybere akci, kterou chce provést, případně nezadá potřebné vstupy.

Výpočet této funkce tedy stojí, ovšem rozhraní musí být schopné reagovat na vstupy od uživatele. Bylo by tedy třeba spouštět nová blokující vlákna na místech, kde se z kódu volá nějaká GUI direktiva, což dále komplikuje i velmi jednoduché programy.

Autor této práce netvrdí, že výše zmíněné problémy jsou neřešitelné a že směrem nastíněným v práci [9] vůbec není možné se vydat. Dále v této kapitole však bude popsán alternativní přístup ke generování GUI, jehož ambicí je být v mnoha směrech jednodušší a praktičtější.

6.3 Akčně orientovaný generátor

Všechny popsané problémy IO-orientovaného generátoru mají v podstatě stejnou příčinu. Touto příčinou je, že funkce uvnitř svých těl obsahují direktivy pro grafické rozhraní. Kdyby uvnitř těl funkcí nebylo žádných (nebo alespoň bylo co nejméně) direktiv pro grafické rozhraní, nebyl by zdrojový kód tak těžko srozumitelný, byl by snáze spravovatelný a nebylo by potřeba jeho běh násilně uměle přerušovat. Tohoto stavu lze dosáhnout, když akce budou reprezentovány samotnými funkcemi. Hodnoty, které po uživateli program požaduje, pak budou prostě parametry těchto funkcí. Díky tomu postačí jedna anotace napsaná před každou funkcí, která má býti akcí v GUI. Tato anotace může specifikovat typy jejích parametrů, typ akce či případné další pokyny pro začlenění do grafického rozhraní. Mějme například funkci, která počítá průměr ze dvou čísel. Její anotovanou verzi si pak lze představit například takto:

```
(defn average ^{:action :button :result :ratio}
  [^{:type :integer} x
   ^{:type :integer} y]
  (/ (+ x y) 2))
```

Jediná direktiva, která se může vyskytovat uvnitř těl funkcí, by sloužila k zpřístupnění nebo znepřístupnění akcí, které mají na spuštěnou akci navazovat. Tyto akce by se pak mohly spolu se svými požadovanými vstupy shlukovat do formulářů například podle toho, které akce jsou společně přístupné, případně které vstupy jsou k jejich spuštění potřeba.

Vzhledem k řešení nakonec zvolenému v této práci jde výše popsaný postup správným směrem. Nakonec je však také zavržen, neboť s ním jsou opět spojeny zásadní problémy. Hlavním problémem je skutečnost, že k tomu, aby bylo možné shlukovat akce do skupin nebo-li formulářů, je nutné ještě před spuštěním programu vědět, které akce jsou v průběhu programu společně přístupné a které naopak spolu nikdy přístupné nejsou. Jde tedy o vytvoření jakéhosi grafu návaznosti jednotlivých akcí. Množina dostupných akcí po spuštění nějaké jiné akce však může záviset na uživatelském vstupu. Triviálním příkladem může být například formulář pro přihlášení uživatele do nějakého informačního systému. Pokud se do systému přihlásí běžný uživatel, jistě mu budou přístupné jiné akce než například administrátorovi. Lze si představit i složitější případ, kdy dostupné akce jsou odvozeny na základě nějakého systému pro přístupová práva. Konkrétní množina dostupných akcí tak může záviset na výsledku procházení nějaké složité datové struktury. Určit tedy, jaké akce jsou v kterém okamžiku dostupné pouze statickou analýzou kódu, může být v obecném případě až nerozhodnutelný problém. Na druhou stranu, různá omezení, která by bylo nutno zavést, aby taková analýza byla použitelná, zase omezují vyjadřovací sílu jazyka, ve kterém je zdrojový kód psán.

Další úskalí výše zmíněného postupu spočívá v tom, že i kdyby se povedlo nastavit systém direktiv tak, aby bylo možné provést statickou analýzu, pro programátora bude velmi složité si představit, jak vlastně výsledné uživatelské rozhraní bude vypadat. Čím bude systém jednodušší, tím těžší bude v něm programovat a naopak, čím větší volnost bude nabízet, tím těžší bude dohlédnout výsledky z takové analýzy vzešlé.

6.4 Koncept FSUI

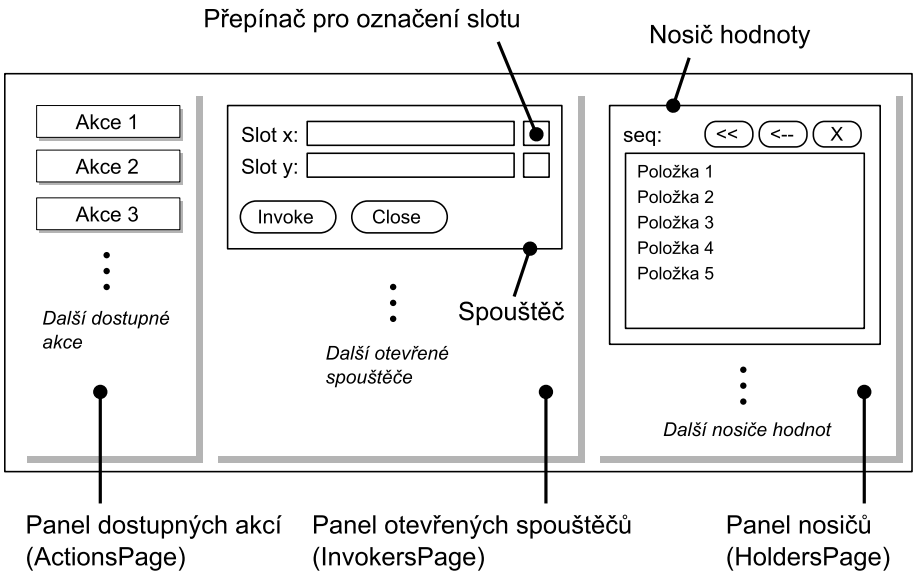
Jestliže výše uvedené přístupy k řešení problému automatického generování GUI se ukazují jako neschůdné nebo příliš komplikované pro možnost praktického využití, nabízí se myšlenka zkusit se na tento problém podívat ze zcela jiného úhlu pohledu. Předešlé náhledy vycházely z představy, že grafické rozhraní business aplikace je v podstatě množství různých formulářů, které se zobrazují podle toho, jakými cestami se ubírá běh ovládaného programu v reakci na uživatelský vstup. Tento pohled na GUI je klasický avšak typický pro v současné době převládající

objektově orientované paradigma. Stejně jako funkcionální programování přináší úplně nový styl do oblasti implementace logiky programu, může i uživatelské rozhraní tento program ovládat jiným než tradičním způsobem a více v souladu s funkcionálními principy.

Zde se nabízí otázka, jaký je vlastně rozdíl v práci s objektově orientovaným a funkcionálním programem z hlediska jeho koncového uživatele, pokud zkusíme abstrahovat od použití konkrétního uživatelského rozhraní. Jelikož základním stavebním kamenem objektově orientovaného programu jsou objekty, práce s programem probíhá tak, že uživatel vytváří či vybírá objekty, na nichž potom volá operace (metody), které vyžadují vytvořit či odněkud vybrat další objekty a tak dále. Naproti tomu základním stavebním kamenem funkcionálních programů jsou funkce. Tedy uživatel analogicky nejprve vybírá funkce, které chce použít a na základě požadovaných parametrů pak vybírá hodnoty, které pomocí vybrané funkce transformuje na hodnoty jiné. Zásadní je však také to, že tyto hodnoty jsou samy o sobě neměnitelné, tedy jen ke čtení. Čili pokud jedna funkce vrátí nějakou hodnotu, například seznam objednávek v elektronickém obchodě, do tohoto sezname nelze "jen tak" přidat novou objednávku. Na přidání nové objednávky program poskytuje funkci, která jako své parametry bere seznam objednávek plus novou objednávku a jejím výstupem pak je nový seznam aktualizovaný o tuto novou objednávku. Z takového pohledu na ovládání funkcionálního programu vychází myšlenka přizpůsobit uživatelské rozhraní těmto principům a vytvořit jakýsi univerzální způsob pro strukturování grafického rozhraní pro funkcionální programy – Functionally Structured User Interface (FSUI).

6.5 Specifikace fungování grafického FSUI

Návrh grafického FSUI je vyobrazen na obrázku 6.3. Grafické FSUI funguje na základě interakce komponent tří různých druhů. Prvním druhem jsou takzvané *akce* (*actions*). Akce představují jednotlivé funkce dostupné uživateli a v GUI jsou reprezentovány tlačítky. Každá akce odpovídá právě jedné funkci ze zdrojového kódu programu. Pokud uživatel spustí nějakou akci zobrazí se druhý typ komponenty jménem *spouštěč* (*invoker*). Tato komponenta obsahuje takzvané *sloty* (*slots*), které odpovídají parametrům funkce příslušné ke spuštěné akci. Každý slot má svůj *typ*, který vymezuje hodnoty, se kterými dokáže vybraná funkce pra-



Obrázek 6.3: Návrh grafického FSUI.

covat. Do těchto slotů může uživatel přiřadit hodnoty, na které chce vybranou funkci aplikovat. Jakmile jsou hodnoty přiřazeny do slotů, funkci je možné spustit. Výsledek funkce se pak zobrazí ve třetím druhu komponenty, což je takzvaný *nosič hodnoty* (*value holder*). Nosiče zobrazují grafické prvky podle typu hodnoty, kterou spuštěná funkce vrátí. Každý ze třech druhů komponent se zobrazuje ve speciálním panelu, či stránce (page) jak ukazuje obrázek 6.3.

6.5.1 Akce

Implementace FSUI v této práci se omezuje na reprezentaci akcí pomocí tlačítek zobrazených pod sebou v levé části hlavního okna aplikace. Program samozřejmě může obsahovat velké množství funkcí a ne všechny jsou použitelné v aktuálním stavu programu. Akce tedy lze zobrazovat či skrývat pomocí direktiv přímo ve zdrojovém kódu. Každá funkce může zpřístupnit či znepřístupnit libovolné množství akcí a tak dávat uživateli najevo, které úkony lze v danou chvíli provádět. Zásadní vlastností akcí je, že přímo nespouštějí příslušné funkce, nýbrž pouze vyvolávají zobrazení spouštěče.

6.5.2 Spouštěče

Na panelu spouštěčů je možné mít spouštěčů zobrazeno libovolné množství a každý z nich lze spustit libovolně mnohokrát stisknutím tlačítka `invoke` nebo jej zavřít tlačítkem `close`. Pomocí spouštěče tak lze volat stejnou funkci vícekrát za sebou s různými hodnotami parametrů. Spouštěč se zobrazí vždy po kliknutí na nějakou akci. Výjimkou je případ, kde funkce příslušná dané akci nemá žádné vstupní parametry. V takovém případě se zobrazí rovnou nosič její návratové hodnoty. Návratová hodnota spouštěče nesmí být nikdy `nil`. Spouštěč vždy musí vracet smysluplnou hodnotu.

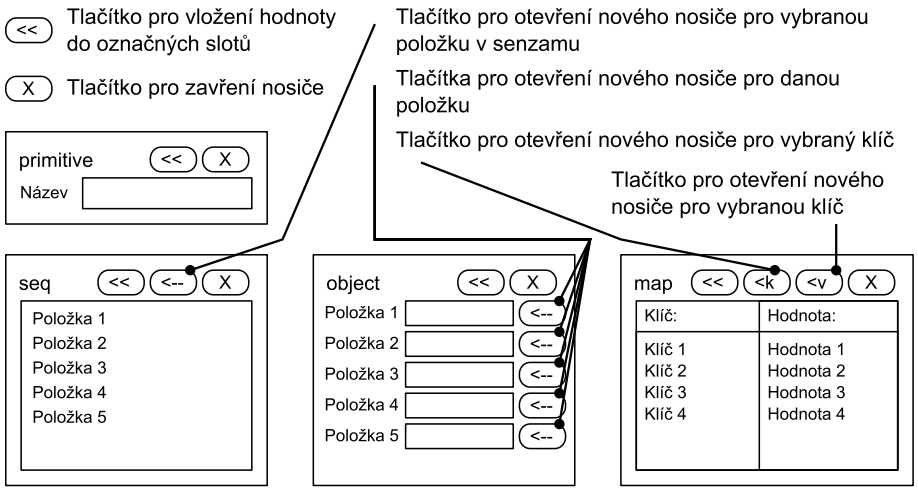
Parametry funkcí se zadávají pomocí slotů. Každý slot odpovídá jednomu parametru funkce. Jde v podstatě o textové políčko, které zobrazuje textovou reprezentaci hodnoty, která do něj byla přiřazena. U primitivních datových typů, jako jsou čísla či řetězce, je možné hodnoty zadávat přímo do políčka. Slot upozorní uživatele, pokud zadává hodnotu v nesprávném formátu a nedovolí mu takovou hodnotu vložit. Složitější struktury, jako jsou například seznamy, se musí do slotu přiřadit z nějakého nosiče hodnoty. K označení slotu jako příjemce hodnoty slouží zaškrtávací tlačítko v jeho pravé části (viz obrázek 6.3). Takto je možno označit více slotů najednou bez ohledu na to, ve kterém jsou zrovna spouštěči. Hodnoty z vybraného nosiče lze pak poslat do více slotů najednou.

Otevřený spouštěč si pamatuje, kterou akcí byl spuštěn. Pokud se tato akce znepřístupní, všechny spouštěče touto akcí otevřené se automaticky zavřou, aby uživatel nemohl provádět akce, které již nejsou dostupné.

6.5.3 Nosiče hodnot

Nový nosič hodnoty je vždy vytvořen po spuštění nějakého spouštěče jako kontejner pro návratovou hodnotu volané funkce. Nosičů existují čtyři typy podle toho, jaký typ hodnoty zrovna představují (viz obrázek 6.4). Typy hodnot ve FSUI se specifikují pomocí typového systému popsaného dále v této kapitole.

Každý nosič hodnoty může být uživatelem zavřen, čímž se jeho hodnota nenávratně ztrácí. Složené nosiče, jako jsou seznamy, mapy a objekty, je možné takzvaně rozebírat. To znamená, že uživatel může například vybrat položku seznamu



Obrázek 6.4: Čtyři druhy spouštěčů pro odpovídající typy hodnot.

v jednom nosiči a stisknutím příslušného tlačítka vytvořit nový nosič obsahující pouze tuto položku. Takto je možné volat funkce na vybrané části různých složených datových struktur.

Nosiče mohou být takzvaně aktivní, což se projeví tak, že tlačítko pro odeslání hodnoty se zpřístupní. Nosič je aktivní tehdy, jestliže typ hodnoty, kterou obsahuje, je podtypem všech typů právě vybraných slotů. Jinými slovy, hodnota z tohoto nosiče může být poslána do všech právě vybraných slotů. Toto odeslání se provede právě zmíněným tlačítkem. Nosič, jehož hodnota nemůže být poslána do všech vybraných slotů, aktivní není a tlačítko pro odeslání hodnoty není přístupné.

Důležitou vlastností nosičů hodnot je, že jsou pouze ke čtení. Hodnoty v nich obsažené nelze v žádném případě nijak měnit. Tato vlastnost může na první pohled vypadat neprakticky. Paradigma funkcionálního programování však jasně říká, že hodnoty se nemění a k vyrábění nových hodnotu slouží funkce. Tento princip ve skutečnosti přináší výhody, neboť programátor může ve zdrojovém kódu jasně definovat, které hodnoty a kdy je možno transformovat v jiné a především jaké struktury je možno v daném okamžiku a stavu aplikace vytvářet. Například webová aplikace implementující internetový obchod jistě nechce dovolit běžným zákazníkům vytvářet objekt představující nový výrobek. K vytvoření nového výrobku slouží funkce, která je dostupná pouze pokud je přihlášen administrátor obchodu. Ten pak může tento nový typ výrobku použitím jiné funkce zařadit do prodeje. Vzniká tak jednoduché pravidlo, kdy každá hodnota, která je přiřazena

nějakému slotu, je buď primitivní, a tedy mohla být vytvořena takříkajíc přímo na místě konkrétního slotu, nebo vznikla jako návratová hodnota funkce spuštěné pomocí nějaké akce.

6.6 Typový systém

Ačkoli je uživatel programu limitován na vytváření hodnot pomocí funkcí, má na druhou stranu značnou svobodu v tom, které funkce spouštět a jaké hodnoty jim předávat skrze sloty. Aby nedocházelo k nedorozuměním a zbytečným chybám, každá funkce by měla jasně deklarovat, hodnoty jakého typu očekává na vstupu a jakého typu je její návratová hodnota. Jazyk Clojure je však dynamicky typovaný a proto je potřeba informace o typech dodat pomocí metadat ve zdrojovém kódu. Na druhou stranu však tento fakt umožňuje vytvořit typový systém téměř nezávislý na typech v Clojure, neboť funkce samy nijak typy svých parametrů neomezují. Implementace konkrétního typového systému by měla splňovat následující požadavky.

1. Typový systém by měl být dostatečně jednoduchý. Pro uživatele je především důležité, aby byl schopen zorientovat se v grafickém rozhraní a poznat, s jakými hodnotami je daná funkce schopna pracovat.

2. Zdaleka není potřeba automatické odvozování typů například jako v jazyce Haskell ani pro funkce ani pro hodnoty. Pro funkce se typy parametrů a návratových hodnot specifikují pomocí metadat přímo v kódu. Co se týče hodnot, jejich typ je vždy znám předem, neboť uživatel může vytvářet nové hodnoty pouze jako návratové hodnoty funkcí nebo pomocí slotu s jasně definovaným typem.

3. V typovém systému by měly být obsaženy primitivní typy a složené struktury, jako jsou například seznamy a mapy. Důležité jsou také záznamy či objekty, které slučují předem definovaný počet položek do jednoho typu. Tak bude možné pracovat s typy jako `zaměstnanec`, `výrobek` apod.

6.6.1 Abstraktní datové struktury

Na samém začátku je třeba vymezit, s jakými datovými strukturami budou aplikace využívající FSUI moci pracovat. Datových struktur je ve funkcionálním i jiném programování samozřejmě nepřeberné množství. Ovšem podobně jako v matematice, kde jsou všechny složitější struktury postaveny na pojmu množina, dají se i datové struktury vyjádřit pomocí poměrně malého počtu určitých elementárních struktur. Tyto elementární struktury jednak odrážejí kromě skutečných potřeb programátorů také způsob, jakým člověk dává řád věcem v reálném světě kolem sebe.

Obecně lze datové struktury dělit na [33]:

- Primitivní (čísla, znaky, řetězce ...)
- Složené

Složené datové struktury slouží k agregaci dalších struktur a často se o nich mluví jako o kolekcích. Kolekcí existují tři hlavní druhy, které korespondují s kolekcemi v jazyce Clojure.

- **Sekvenční**, založené na matematickém pojmu n-tice. Mají definované pořadí prvků a může se v nich vyskytovat více stejných prvků. Patří sem seznamy, vektory, pole apod.
- **Množinové**, založené na pojmu množina. Prvky nemají jasně definované pořadí a nemohou se opakovat.
- **Asociativní**, založené na pojmu zobrazení. Každý prvek má přiřazen klíč, pomocí kterého je jednoznačně identifikován.

Hovoří-li se o asociativních strukturách, je možné z nich vydělit významnou podskupinu, která představuje struktury s pevným, předem daným pořadím a počtem prvků. Klíče jsou pak jednoduchá jména (například klíčová slova v jazyce Clojure). Takové struktury se často nazývají záznamy, případně objekty a jejich prvky atributy, vlastnosti apod.

Většina programovacích jazyků včetně Clojure implementuje všechny výše popsané typy struktur buď přímo nebo pomocí dodatečných knihoven. Tyto struk-

tury tedy tvoří jakýsi základní způsob organizace dat. Výše popsané typy navíc abstrahují od konkrétních implementací, které se mohou zásadně lišit svojí efektivitou v nejrůznějších situacích. Proto je rozumné založit typový systém právě na těchto abstraktních typech.

6.6.2 Specifikace typového systému

Každá funkce, která je spustitelná jako akce v GUI, musí být opatřena takzvanou *typovou signaturou*. Tato signatura je seznam *typových forem*. Každá typová forma popisuje typ hodnoty. První forma v signatuře udává typ návratové hodnoty a ostatní formy udávají typy parametrů funkce. Typové formy i signatury jsou korektní S-výrazy jazyka Clojure. Například signatura funkce, která vrátí součet celých čísel v seznamu, vypadá takto

```
(:int (seq :int))
```

Každá funkce, která komunikuje s uživatelem, tak jasně deklaruje, s jakým typem hodnot je schopna pracovat. Samotná funkce pak také musí být na všechny tyto hodnoty připravena. Proto je třeba jasně definovat, jaké hodnoty jsou pro ten který typ přípustné. Tím se zabývá právě následující část tohoto textu.

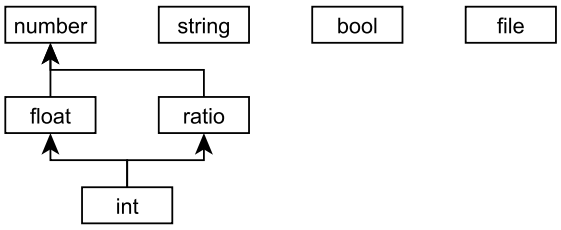
Všechny typové formy a signatury odpovídají gramatice ve výpisu 6.2. Jak vidno, typový systém zahrnuje pouze tři složené datové typy `seq`, `map` a `object`. Typ `seq` zahrnuje většinu sekvencí, u kterých lze prvky seřadit za sebe. Jsou to přesně seznamy, vektory a množiny z jazyka Clojure, tedy právě ty hodnoty, které splňují predikáty `list?`, `vector?` a `set?` [27]. K tomuto sjednocení sekvenčních a množinových typů dochází proto, že z pohledu uživatele mezi nimi není zásadního rozdílu. To, že v množinách se prvky nemohou opakovat, uživatel nijak nepocítí, neboť množiny, stejně jako všechny ostatní hodnoty, lze vytvářet pouze pomocí funkcí. Funkce, která vytváří množinu, se tak sama stará o to, aby uživateli nedovolila zadat více stejných hodnot. Toto rozhodnutí tedy přispívá k jednoduchosti typového systému.

Mapy jsou z pohledu uživatele v podstatě tabulky se dvěma sloupci – klíč a hodnota. Typu `map` vyhovují všechny hodnoty odpovídající predikátu `map?` [27]. Posledním složeným typem je `object`, který představuje objekt nebo-li posloup-

Výpis 6.2 Struktura typové formy.

```
<signature> ::= (<form> <form-list>)
<form-list> ::= <form> | <form> <form-list>

<form>      ::= <prim> |
                (seq <form>) |
                (map <form> <form>) |
                (object <name>)
<prim>      ::= :bool | :int | :float | :ratio |
                :number | :string | :file
<name>      ::= libovolný symbol začínající malým písmenem
```



Obrázek 6.5: Hierarchie primitivních typů.

nost atributů. Tato typová forma obsahuje pouze jméno tohoto objektu. Samotná definice objektu se provede pomocí anotací ve zdrojovém kódu, ze kterého se rozhraní generuje. Způsob definice objektů bude popsán v části zabývající se systémem metadat pro anotaci zdrojového kódu. Zde je nutné zmínit pouze to, že typu `object` vyhovují pouze instance datových typů vytvořených v Clojure pomocí makra `defrecord` [27].

Primitivní datové typy odpovídají primitivním typům v jazyce Clojure, tedy nemají žádná omezení na velikost, počet desetinných míst apod. Jsou uspořádány do IS-A hierarchie, kterou ukazuje obrázek 6.5. Je tedy možné použít například celé číslo tam, kde je očekáván zlomek apod. Typ `file` odpovídá třídě `java.io.File` a slouží k práci se soubory.

Pro korektní fungování celého typového systému je potřeba také jasně definovat, kdy je možné předat funkci jako parametr hodnotu, jejíž typ se od deklarova-

ného typu tohoto parametru liší. Intuitivně je zřejmé, že například funkce, která jako svůj vstup požaduje seznam desetinných čísel, bude umět pracovat také se seznamem celých čísel. Tento vztah zde bude zapisován takto

$$\texttt{(seq :int)} \preceq \texttt{(seq :float)}$$

Relaci $\preceq$ nazvěme *podtyp*. Její význam je takový, že pokud pro dva typy T_1 a T_2 platí $T_1 \preceq T_2$, potom lze hodnotu typu T_1 použít všude tam, kde je očekávána hodnota typu T_2.

Definice relace podtyp

Označme $\mathcal{T}$ množinu všech korektních typových forem. Aby bylo možno definovat relaci podtyp, je potřeba zavést relaci $\preceq_p \subset \mathcal{T} \times \mathcal{T}$ nazvanou *primitivní podtyp*. Tato relace je částečné uspořádání nad primitivními typy zobrazené na obrázku 6.5. Pokud alespoň jeden z typů T_1, T_2 není primitivní, pak jsou T_1 a T_2 v uspořádání $\preceq_p$ neporovnatelné.

Relaci $\preceq$ definujeme induktivně. Nechť $\preceq \subset \mathcal{T} \times \mathcal{T}$ a platí:

1. $\forall T \in \mathcal{T} \quad T \preceq T$
2. $\forall T_1, T_2 \in \mathcal{T} \quad T_1 \preceq_p T_2 \Rightarrow T_1 \preceq T_2$
3. $\forall T_1, T_2 \in \mathcal{T} \quad T_1 \preceq T_2 \Rightarrow \texttt{(seq } T_1\texttt{)} \preceq \texttt{(seq } T_2\texttt{)}$
4. $\forall T_1, T_2, T_3, T_4 \in \mathcal{T} \quad T_1 \preceq T_2 \wedge T_3 \preceq T_4 \Rightarrow \texttt{(map } T_1\ T_2\texttt{)} \preceq \texttt{(map } T_3\ T_4\texttt{)}$

Definice přímo říká, že relace podtyp je reflexivní. Tranzitivita se dokáže indukcí, která opět přímo plyne z definice. Relace podtyp tedy tvoří částečné uspořádání nad množinou všech typových forem.

6.6.3 Rozšíření typového systému

Výše popsaný typový systém má samozřejmě kvůli svojí jednoduchosti mnoho omezení. Zdaleka nedokáže přiřadit typ každé hodnotě, která může být v jazyce Clojure vytvořena, ani vyjádřit signaturu libovolné funkce, kterou je možné v Clojure napsat. Mezi takové hodnoty či funkce patří například heterogenní

sekvence, seznamy seznamů zanořené do libovolné hloubky, funkce pracující se seznamy libovolného typu či funkce vyšších řádů. V disertační práci [34], která implementuje statické typové odvozování pro Common Lisp, lze nalézt inspiraci, jak by se dal zde představený typový systém rozšířit, aby uměl specifikovat typy širší škály hodnot. Zda však taková rozšíření přinášejí pro koncové uživatele a programátory nějaké zásadní výhody, je otázkou do následující diskuze.

Typové proměnné

Lákavé je především zavedení typových proměnných, bez kterých není možné jednou signaturou popsat typ například funkce `first`, která vrací první prvek sekvence. Její signatura by s použitím typových proměnných vydala takto

```
(A (seq A))
```

Pokud by tato funkce dostala jako svůj parametr například sekvenci celých čísel, substituční algoritmus by odvodil, že funkce vrací hodnotu typu `int`. Typové proměnné na první pohled vypadají velmi užitečně. Proti jejich zavedení však mluví důvody jak konceptuální tak technické.

Co se týče hlediska konceptuálního, je dobré připomenout, jakým způsobem program komunikuje s uživatelem. Funkce v programu je možné dělit na ty, které jsou zobrazeny jako akce v GUI a tedy přímo vykonávají vůli uživatele a na ty, které vykonávají samotnou logiku aplikace a uživateli jsou proto skryty. Typové proměnné by byly důležité především pokud by bylo třeba specifikovat signatury pro funkce z druhé skupiny, neboť to jsou funkce, které často vyžadují vysokou míru abstrakce a volnosti v tom, s jakými hodnotami pracují. Příkladem budiž například funkce, která vrátí všechny položky se sudým pořadím v nějaké sekvenci.

Naopak funkce, které komunikují s uživatelem, jsou typově co nejkonkrétnější, nepracují s obecnými seznamy nýbrž se seznamy zboží, zaměstnanců, objednávek apod. Správně tedy funkce, které by využily typových proměnných, vůbec typové signatury nepotřebují, neboť vykonávají samotnou logiku aplikace.

Co se týká hlediska technického, není například zřejmé, jak přesně by měl pracovat substituční algoritmus. Mějme například funkci, která má signaturu

```
((seq A) (seq A) (seq A))
```

Z takové signatury není zřejmé, jaká substituce se očekává za proměnnou `A`. Pokud funkce například dostane dva seznamy řetězců (`string`), je substituce jasná. Když ale například obdrží seznam desetinných čísel (`float`) a seznam zlomků (`ratio`), není jednoznačné, seznam jakého typu má vrátit. Je možné říct, že typy (`float`) a (`ratio`) jsou relací podtyp neporovnatelné, a tudíž takto funkci vůbec zavolat nelze. Také je ale možný přístup takový, že za proměnnou se dosadí nejmenší společný nadtyp typů (`float`) a (`ratio`). Funkce tak vrátí seznam obecných čísel (`numder`). Otázkou je, zda funkce s podobnou signaturou vždy takovéto chování předpokládá a umí s ním vždy pracovat. Tím vyvstává například otázka, zda nezavést syntaxi pro nějaké omezování možných hodnot typových proměnných. To se však již dostáváme daleko od praktické použitelnosti pro koncové uživatele, jimž je grafické rozhraní určeno.

Rekurzivní typy

Jedním z dalších omezení výše specifikovaného typového systému je nemožnost vyjádřit typ například libovolně do sebe zanořených seznamů celých čísel. Vycházejíc z práce [34] je možné si pro takový typ představit například formu

```
(rec typ1 (seq [:int typ1])),
```

kde `typ1` je jméno typu a vektorem se zapisuje výběr z více možností. Výhoda takovýchto typů by mohla spočívat například ve snadné definici různých stromových struktur. Vnořené struktury je však také možné definovat jako typ `object`. Jedna nebo i více z jeho položek může bez problémů mít také typ `object` a tímto způsobem vytvořit libovolně zanořenou strukturu. Výhodou je, že takto vytvořené typy jsou jasně pojmenované, takže v GUI lze pak pracovat například s libovolně zanořenými kategoriemi výrobků namísto s libovolně zanořenými sekvencemi. Z těchto důvodu rekurzivní typy také nejsou do typového systému zařazeny.

Autor této práce však nevylučuje, že v budoucnu pod tíhou praktických požadavků vývojářů může potřeba zavedení nějakého druhu typových proměnných nebo jiných rozšíření nastat. Typový systém proto bude implementován tak, aby jej bylo možné snadno rozšiřovat o nové syntaktické koncepty nebo další primitivní typy.

6.7 Generátor FSUI

Tato sekce popisuje fungování generátoru FSUI, především tvar jeho vstupu a výstupu. Co se týče vstupu, je třeba detailně popsat systém metadat pro anotování zdrojového kódu. Výhodou funkcionálně strukturovaného uživatelského rozhraní je, že není potřeba generovat téměř žádný kód. Grafické rozhraní totiž funguje na základě grafické reprezentace konkrétní instance datového modelu FSUI. Jediné, co musí generátor analyzující zdrojový kód vygenerovat, je popis všech akcí a objektů, se kterými potom pracuje implementace uživatelského rozhraní.

6.7.1 Systém metadat

Vstupem pro generátor FSUI je zdrojový kód programu napsaný v jazyce Clojure opatřený metadaty, která popisují jeho interakci s uživatelem. Systém metadat pro anotování zdrojového kódu je relativně jednoduchý a přímočarý. Základem jsou makra `defobject` a `defaction`. První z nich slouží k definici objektu, který pak bude možné v signaturách používat jako `(object <typ>)`. Typ, který představuje komplexní číslo, pak může vypadat například takto

```
(defobject complex [x :float
                    y :float])
```

Makro `defobject` jako svůj první parametr očekává symbol, který představuje jméno objektu. Druhý parametr pak specifikuje atributy tohoto objektu. Zde se očekává vektor, ve kterém se střídají dvojice symbolů a jejich typů. Jména symbolů musí odpovídat jménům klíčů v datové struktuře vytvořené pomocí makra `defrecord`, která tvoří hodnotu tohoto typu.

Makro `defaction` slouží k vytvoření funkce, která bude zároveň akcí ve výsledném grafickém rozhraní. Jako svůj první parametr očekává signaturu funkce, kterou definuje. Dále je jeho syntaxe totožná se syntaxí makra `defn` popsaného v kapitole 5 (podrobněji v [27]). Makro `defaction` tak vytváří obyčejnou funkci, která má stejné jméno, jako je jméno akce. S touto funkcí lze pak mimo grafické rozhraní pracovat jako s běžnou funkcí v jazyce Clojure. Příklad akce, která počítá absolutní hodnotu komplexního čísla, pak vypadá takto

```
(defaction (:float (object complex))
  complex-abs [{x :x y :y}]
  (Math/sqrt (+ (* x x) (* y y)))
```

Kromě těchto metadat je pak možné uvnitř libovolné funkce (nemusí to být akce) volat následující tři direktivy pro zpřístupňování akcí v GUI. Všechny jako svůj první parametr očekávají libovolnou hodnotu, která je po jejich provedení vrácena. Mohou tedy být použity na libovolném místě v kódu, neboť se chovají jako identita.

1. `@update-actions` – slouží k zpřístupnění a znepřístupnění konkrétních akcí. Použití může vypadat takto

   ```
   (@update-actions <návratová hodnota>
                    :enable  [:akce1 :akce2]
                    :disable [:akce3])
   ```

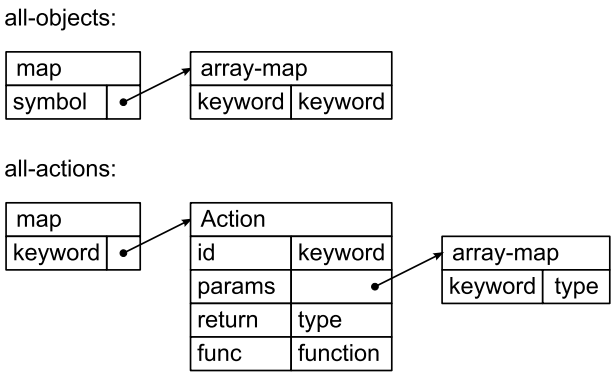
   Akce označené jako `:enable` se zpřístupní a akce označené jako `:disable` znepřísutpní.

2. `@enable-actions-only` očekává libovolný počet akcí. Tyto jsou pak v grafickém rozhraní zpřístupněny, všechny ostatní akce jsou naopak znepřístupněny. Příklad použití

   ```
   (@enable-actions-only <návratová hodnota> :akce1 :akce2)
   ```

3. `@disable-actions-only` má stejnou syntaxi a pracuje přesně opačně jako předchozí direktiva. Předané akce jsou v grafickém rozhraní znepřístupněny, všechny ostatní akce jsou naopak zpřístupněny.

Všechny tyto direktivy začínají znakem `@`, z čehož je zřejmé, že jde o volání funkce uložené v nějakém synchronizačním primitivu – v tomto případě atomu. Při spouštění anotovaného programu bez grafického rozhraní tyto funkce nemají žádný vedlejší efekt a nijak neovlivňují běh programu. Pokud se však spustí vygenerované grafické rozhraní, při jeho inicializaci se implementace těchto funkcí změní na takovou, která správně nastaví přístupné akce v modelu GUI.



Obrázek 6.6: Výstupní struktury generátoru.

6.7.2 Výstup generátoru

Výstupem generátoru je soubor obsahující kód opět v jazyce Clojure. V tomto kódu jsou vytvořeny datové struktury, které obsahují popis všech akcí a objektů používaných ve vstupním zdrojovém kódu. Dále je zde obsažena metoda `-main`, která spouští uživatelské rozhraní a tím i celý program. Popis akcí a objektů je uložen v mapách `all-actions` a `all-objects`. Strukturu těchto dvou map ukazuje obrázek 6.6.

Mapa `all-actions` mapuje identifikátory akcí na jejich skutečné instance. Každá akce sestává ze čtyř položek. Položka `id` slouží jako jedinečný identifikátor akce. Mapa `params` zobrazuje názvy parametrů akce na jejich typy. Položka `return` je typ návratové hodnoty akce a nakonec `func` je funkce, která akci představuje. Příklad mapy obsahující záznam pro výše uvedenou ukázkovou akci počítající absolutní hodnotu komplexního čísla vypadá takto

```
(def all-actions
  {:complex-abs (Action. :complex-abs
                         (array-map :c '(object complex))
                         :float
                         complex-abs)})
```

Pro mapu parametrů je použita takzvaná array-mapa, která je v Clojure implementována jako pole dvojic klíč-hodnota. Díky tomu zůstává zachováno pořadí jejích položek.

Mapa `all-objects` mapuje jména objektů v podobě symbolů na výčet jejich atributů. Tento výčet je stejně koncipovaná array-mapa, jako ta, která ja použitá pro výčet parametrů akce. Mapa objektů obsahující záznam pro komplexní číslo vypadá takto

```
(def all-objects
  {'complex (array-map :x :float :y :float)})
```

Méně triviální příklad vstupu a výstupu pro generátor FSUI je součástí kapitoly 8 věnující se případové studii. Ta ukazuje možnosti konceptu FSUI a použití výše specifikovaného generátoru do větší hloubky.

Kapitola 7

Framework pro generování GUI

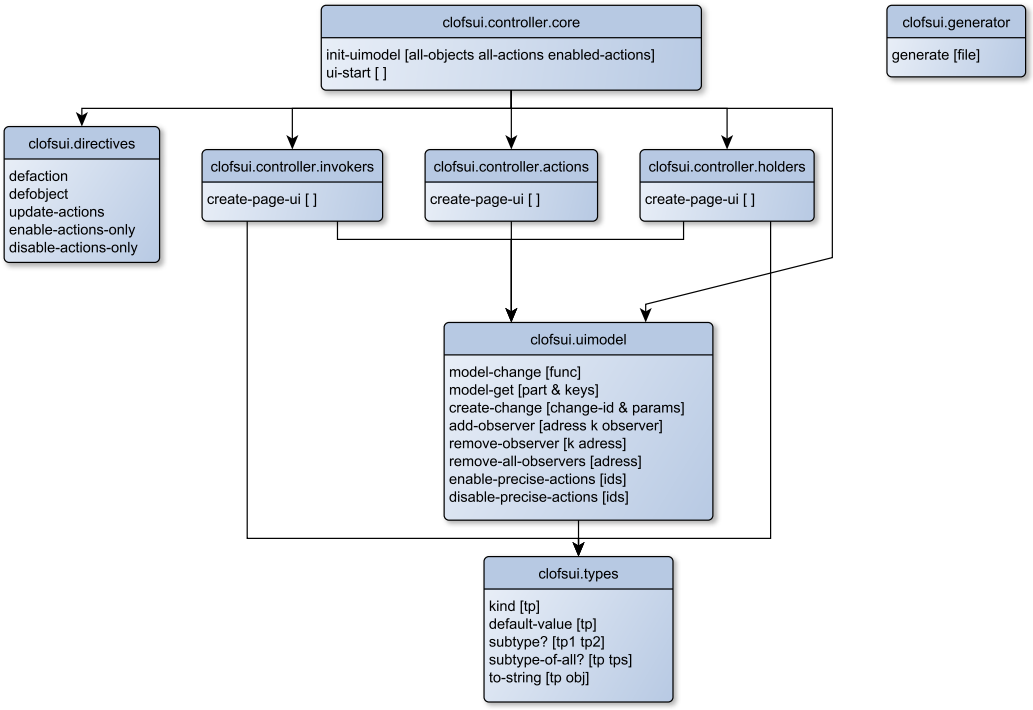
Následující kapitola se zabývá implementací frameworku, který v souladu s cílem práce umožňuje generování grafického uživatelského rozhraní podle principů FSUI z anotovaného kódu napsaného v jazyce Clojure. Tento framework má dvě oddělené části. První z nich je samotný generátor, který prochází anotovaný kód a produkuje popisy akcí a objektů. Z hlediska množství kódu a implementační náročnosti je však významnější část druhá, tedy samotné funkcionálně orientované uživatelské rozhraní, které ovládá vstupní program. Jeho implementace je netriviální především kvůli tomu, že principy, na kterých je založeno grafické uživatelské rozhraní, jdou často proti principům funkcionálního programování.

Veškerá funkcionalita je implementována v jazyce Clojure verze 1.3.0. V tomto jazyce je zvykem členit kód do jmenných prostorů (namespaces), které jsou podobné například balíčkům z jazyka Java. Každý jmenný prostor představuje jeden zapouzdřený modul. Každý modul publikuje nějaké veřejné funkce, které potom mohou ostatní moduly volat. Jednotlivé moduly, jejich vzájemné závislosti a veřejná rozhraní zobrazuje obrázek 7.1.

7.1 Implementace generátoru FSUI

Generátor FSUI je implementován ve jmenném prostoru

`clofsui.generator`



Obrázek 7.1: Hierarchie modulů v systému FSUI.

Implementace je velmi přímočará a vejde se do jednoho souboru o přibližně 100 řádcích včetně komentářů. Pro srozumitelnost myšlenky zatím tento generátor umí pracovat jen s jedním souborem na vstupu. Kód veškerých akcí a definice objektů musí tedy být obsaženy v jednom zdrojovém souboru. Toto omezení lze samozřejmě rozšířením generátoru v budoucnu odstranit.

Jedinou veřejnou funkcí generátoru je funkce `generate`, která jako svůj vstup očekává Clojure soubor, jenž obsahuje anotovaný kód pro grafické rozhraní. Generátor funguje tak, že lineárně projde seznam všech Clojure forem ve vstupním souboru, vyčlení z nich ty, které jsou důležité pro grafické rozhraní, tedy především volání maker `defaction` a `defobject`. Tyto potom převede do výstupního formátu, jenž byl podrobně popsán v předešlé kapitole. Zopakujme, že jde o mapy akcí a objektů spolu s funkcí `-main`, jejímž zavoláním se spustí grafické rozhraní. Konkrétní implementační detaily tohoto postupu jsou technické záležitosti, jimž je možné porozumět přímo ze zdrojového kódu generátoru a z komentářů v něm. Konkrétní příklad použití generátoru bude popsán v kapitole 8 věnující se případové studii.

Funkce	Popis
`subtype? [tp1 tp2]`	Vrací `true` pokud je typ `tp1` podtypem typu `tp2`.
`subtype-of-all? [tp tps]`	Vrací `true` pokud je typ `tp` podtypem všech typů v sekvenci `tps`.
`default-value [tp]`	Vrací výchozí hodnotu pro zadaný typ.
`to-string [tp obj]`	Převádí hodnotu `obj` typu `tp` na textový řetězec.

Tabulka 7.1: Popis veřejných funkcí v modulu `clofsui.types`.

7.2 Implementace FSUI

Nyní následuje popis implementace samotného funkcionálně strukturovaného uživatelského rozhraní podle specifikace uvedené v předchozí kapitole. Je nutné implementovat datový model uživatelského rozhraní, samotné grafické rozhraní pomocí knihovny Swing [37] a také typový systém.

7.2.1 Implementace typového systému

Implementace typového systému se nachází v modulu `clofsui.types`. Tento modul poskytuje veřejné funkce popsané v tabulce 7.1. Funkce `subtype-of-all?` se používá především ve chvíli, kdy uživatel v grafickém rozhraní vybere několik slotů a je potřeba zpřístupnit pouze ty nosiče, které obsahují hodnoty, jejichž typ je podtypem všech typů ve vybraných slotech. Funkce `to-string` slouží hlavně k zobrazení hodnoty v textovém políčku slotu.

Vzhledem k tomu, že syntaxe jazyka Clojure má blízko k matematickému vyjadřování, je implementace funkce `subtype?` v podstatě přímým přepisem definice relace $\preceq$. Implementace ostatních funkcí je opět velmi přímočará a srozumitelná přímo ze zdrojového kódu.

7.2.2 Doménová logika FSUI

Při implementaci datového modelu FSUI vyvstává problém jak zkombinovat odlišné povahy funkcionálního programu a grafického rozhraní. Funkcionální program se snaží být co nejméně závislý na stavu, kdežto grafické rozhraní je ze své podstaty stavové. V imperativním programování je v této situaci běžně používaným architektonickým vzorem model-view-controller (MVC) [35]. V případě FSUI grafické rozhraní představuje pohled i kontrolní logiku a modelem jsou datové struktury jazyka Clojure. Tyto struktury jsou však z podstaty neměnné a proto nemůže například zaškrtnutí políčka v GUI změnit jen některou položku datové struktury. Taková struktura se musí změnit celá. Překreslovat ale celé uživatelské rozhraní při každém zaškrtnutí políčka je velmi neefektivní.

Při řešení tohoto problému velmi napomáhá sám jazyk Clojure, který je díky své softwarové transakční paměti a synchronizačním primitivům připraven na efektivní správu stavů a identit. Je tedy možné využít referencí, které umí pomocí transakcí synchronizovat více závislých identit. Díky tomu je možné rozdělit model na více synchronizovaných částí. V tomto procesu je ale třeba najít určitou rozumnou míru, neboť úplné rozdrobení datového modelu do referencí zcela popírá funkcionální paradigma a program se tak stává v podstatě imperativním.

Pro správné fungování GUI je nutné, aby každý grafický prvek jako tlačítko či textové políčko věděl, která část modelu mu přísluší a byl tak schopen reagovat na situaci, kdy se nový model v této části liší od starého (nebo-li vznikne nový model se změnou na tomto místě). Je tedy nutné vyvinout systém, který bude jednak schopen adresovat jednotlivé části modelu a jednak informovat jednotlivé grafické prvky, aby reflektovaly změnu modelu.

7.2.3 Implementace modelu FSUI

Model FSUI je implementován v modulu

`clofsui.uimodel`

Veřejné funkce tohoto modulu představují abstraktní rozhraní pro práci s modelem grafického rozhraní. Aby byla práce s modelem co nejjednodušší, `uimodel`

nabízí pouze dvě funkce, které s modelem pracují přímo. Jsou to `model-get` a `model-change`. První dokáže na základě adresy vrátit libovolnou podčást modelu. Druhá slouží jako obecný způsob, jak provést v modelu nějakou změnu a informovat o ní své okolí (např. grafické rozhraní). Obě tyto funkce pracují s modelem uloženým v globálním symbolu, který však mimo `uimodel` není přístupný. Tím pádem má model jednu jedinou centrální instanci, ke které však lze přistupovat pouze použitím funkcí `model-get` a `model-change`. Globální přístupnost je nutná především proto, aby mohly fungovat direktivy pro zpřístupňování akcí ve vstupním programu, které rozhodně nemohou obdržet instanci modelu jako svůj parametr.

Funkce `model-get`

Funkce `model-get` vrátí libovolnou část modelu na základě její adresy. Model je hierarchicky strukturavaný tak, že každá položka má ve svém kontejneru unikátní identifikátor, jímž je nějaké klíčové slovo. Lze tak například zjistit hodnotu jednoho konkrétního slotu takto

```
(model-get :invokers-page :invokers :complex-abs27 :x :value)
```

Funkce `model-change`

Funkce `model-change` jako svůj parametr očekává bezparametrickou funkci, která jak svůj výsledek vrací seznam adres v modelu. Tuto funkci poté spustí uvnitř transakce. Díky tomu lze uvnitř takto spuštěné funkce volat `ref-set` a `alter`, které mohou měnit reference v modelu FSUI. Takto je možné provádět libovolné změny v modelu. Následně jsou pro všechny vrácené adresy zavoláni případní registrovaní posluchači. Posluchači jsou funkce, které mohou o provedených změnách informovat konkrétní části grafického rozhraní.

Funkce `create-change`

Funkce `create-change` slouží jako obecný nástroj pro vytváření často používaných změn. Ve skutečnosti jsou takto vyráběny všechny změny modelu, neboť

ID změny	**Popis**
init	Inicializace modelu UI strukturami z výstupu generátoru.
execute-action	Spuštění akce. Znamená otevření spouštěče s příslušnou funkcí nebo rovnou nosiče s návratovou hodnotou.
select-slot	Zahrnutí slotu do výběru.
new-slot-value	Změna hodnoty slotu.
invoke	Spuštění spouštěče. Znamená otevření nosiče s návratovou hodnotou a aktualizaci dostupných akcí.
close-invoker	Zavření spouštěče.
open-holder	Otevření nosiče pro konkrétní hodnotu.
send-value	Odeslání hodnoty do všech vybraných slotů.
close-holder	Zavření nosiče.

Tabulka 7.2: Seznam změn vytvářených funkcí `create-change`.

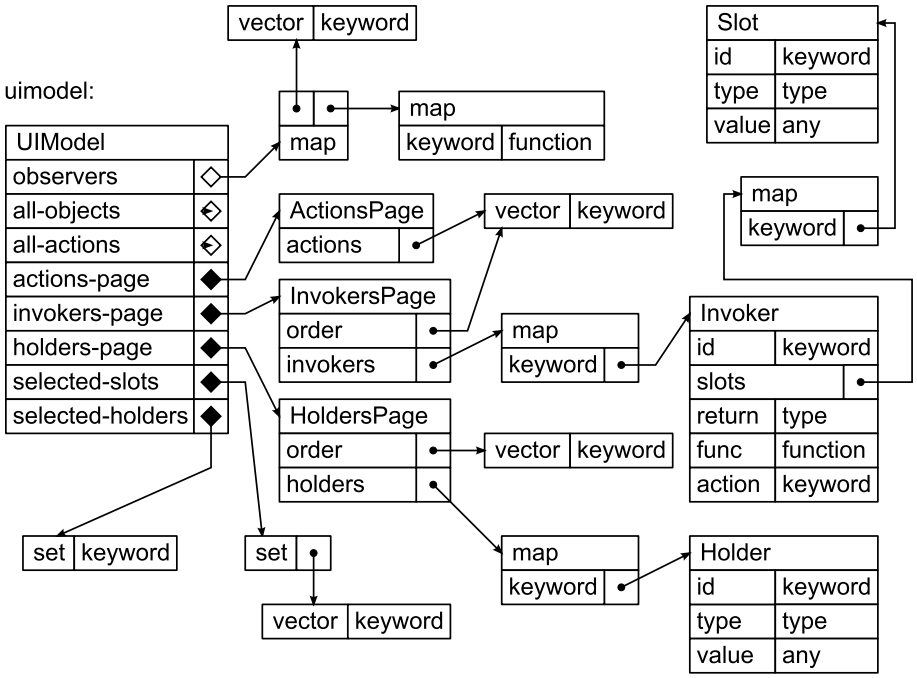
počet používaných operací nad modelem není velký. Funkce `create-change` jako svůj první parametr bere identifikátor změny, což je klíčové slovo. Následuje proměnný počet parametrů, jejichž počet a typ závisí na konkrétní změně. Návratovou hodnotou je funkce, kterou lze rovnou předat funkci `model-change`. Například zavření spouštěče je pak možné provést takto

```
(model-change (create-change :close-invoker :complex-abs27))
```

Všech změn, které slouží ke kompletní manipulaci s modelem, je pouze devět. Jejich stručný popis ukazuje tabulka 7.2. Podrobnější popis těchto změn a technické detaily implementace je možné najít ve zdrojovém kódu.

Funkce `add-observer` a `remove-observer`

Tyto funkce slouží k přidávání a odebírání takzvaných *posluchačů*. Posluchač je funkce, která očekává jeden parametr, kterým je adresa. Tato funkce je pak zavolána pokaždé, když se model změní právě na této adrese. Posluchače je však nejdříve nutno zaregistrovat pomocí funkce `add-observer`, která navíc umožňuje



Obrázek 7.2: Diagram struktury modelu FSUI.

přidat k jedné adrese více různých posluchačů s rozdílným klíčem. Pomocí funkce `remove-observer` je pak možné takového posluchače odstranit.

7.2.4 Struktura modelu

Pod abstraktní vrstvou umožňující přístup k modelu a provádění změn leží konkrétní datová struktura, která tento model implementuje. Její detailní popis ukazuje obrázek 7.2. Prázdné kosočtverce v tomto diagramu představují atomy, plné pak reference. Struktury `all-actions` a `all-objects` jsou již popsány diagramem na obrázku 6.6, proto z těchto položek nevedou žádné šipky.

Modelem celého FSUI je datový typ `UIModel`. Ten obsahuje osm položek, které jsou díky synchronizačním primitivům měnitelné. To, že jsou první tři položky atomy znamená, že jejich změny jsou nezávislé na zbytku modelu. Zbylých pět položek jsou reference, čili jejich změny mohou na sobě být závislé. Tyto změny je třeba synchronizovat skrze transakce, aby model zůstal v konzistentním stavu.

Položka `observers` obsahuje mapu všech posluchačů. Jejími klíči jsou adresy, které adresují jednotlivé části modelu. Hodnotami jsou mapy, které mapují klíče posluchačů na samotné posluchače. Pří startu aplikace jsou výstupní struktury

generátoru `all-objects` a `all-actions` přiřazeny právě do příslušných položek v modelu aplikace a dále se za celý běh aplikace nemění.

Položka `actions-page` udržuje vektor identifikátorů právě aktivních akcí. Spouštěče a nosiče hodnot jsou spravovány stránkami v položkách `invokers-page` a `holders-page`. Obě obsahují mapy jednotlivých spouštěčů a nosičů. Klíčem v této mapě je vždy identifikátor nosiče nebo spouštěče. Obecně ovšem mapy nezachovávají vkládací pořadí prvků, je proto nutné mít ještě vektor `order`, který udržuje správné pořadí jednotlivých identifikátorů. Samotný spouštěč obsahuje kromě identifikátoru (`id`), typu návratové hodnoty (`return`) a funkce, kterou má spouštět (`func`), ještě mapu slotů a identifikátor akce, kterou byl spouštěč vyvolán. Každý slot, stejně jako nosič, obsahuje kromě svého identifikátoru také typ a hodnotu.

Položka `selected-slots` obsahuje adresy právě vybraných slotů. Adresa slotu je vektor obsahující dva prvky – identifikátor spouštěče a identifikátor samotného slotu. Poslední položka modelu grafického rozhraní `active-holders` obsahuje identifikátory aktivních nosičů.

Adresování

Diagram na obrázku 7.2 zároveň ukazuje, jak vypadají adresy jednotlivých částí modelu. Prvky adresy jsou klíče v jednotlivých asociativních strukturách, tedy buď názvy položek datových typů[1] nebo identifikátory jednotlivých komponent. Takovou adresu je pak možné předat funkci `model-get` a získat tak kýženou část modelu. Identifikátory pro spouštěče a nosiče se generují náhodně při jejich vytváření. Identifikátory slotů jsou názvy odpovídajících parametrů funkce, kterou spouštěč volá. Získat tak například hodnotu nějakého nosiče je možné zavoláním funkce

```
(model-get :holders-page :holders :holder271 :value)
```

[1] Připomeňme, že datové typy se také chovají jako asociativní struktury.

7.3 Grafické rozhraní

Implementace vizuální části grafického rozhraní využívá knihovnu Seesaw [36], která umožňuje v jazyce Clojure poměrně pohodlně pracovat s klasickou knihovnou pro tvorbu GUI jménem Swing z jazyka Java [37]. Knihovna Seesaw umožňuje grafická rozhraní psát více funkcionálním způsobem. Tento funkcionální kód je navíc daleko kompaktnější a srozumitelnější než rovnocenné GUI napsané přímo v jazyce Java.

Vizuální část grafického rozhraní je implementována ve čtyřech modulech.

`colfsui.controller.core`
`colfsui.controller.actions`
`colfsui.controller.holders`
`colfsui.controller.invokers`

První z nich poskytuje funkce `init-uimodel` a `ui-start`. První slouží k nastavení položek `all-objects` a `all-actions` v modelu grafického rozhraní a k nastavení akcí přístupných při startu programu. Druhá funkce pak přímo spouští grafické rozhraní. Obě tyto funkce jsou volány z vygenerované metody `-main` ve výstupním souboru generátoru a slouží tak ke spuštění celé aplikace.

Každý ze zbylých tří modulů se stará o zobrazování jedné části FSUI – panelu akcí, panelu spouštěčů a panelu nosičů hodnot. Celé grafické rozhraní funguje na principu posluchačů, kteří reagují na změny v datovém modelu rozhraní a aktualizují příslušné vizuální prvky. Detailnější popis implementace je opět velmi technickou záležitostí závisící navíc na znalosti knihovny Seesaw. Podrobnější popis fungování této vrstvy je tedy opět dobré hledat přímo ve zdrojovém kódu a komentářích v něm.

Kapitola 8

Případová studie

Tato kapitola ukazuje použití implementovaného frameworku pro generování grafického uživatelského rozhraní na příkladu menší aplikace, která demonstruje většinu možností, které tento framework nabízí. Zadáním v této studii je vytvořit malý informační systém, která umožní spravovat městskou knihovnu.

8.1 Specifikace IS knihovny

Informační systém pro správu knihovny bude desktopová aplikace běžící na počítačích v knihovně, kde se k ní budou moci přihlásit jednotliví čtenáři a půjčovat si knížky. Program bude spravovat databázi knížek, se kterou budou moci administrátoři systému pracovat – přidávat nové knihy, registrovat nové čtenáře apod.

Systém po spuštění zobrazí přihlašovací formulář, kde se budou moci uživatelé přihlásit pomocí jména a hesla. Uživatelé jsou dvojího typu – čtenáři a administrátoři. Pokud se do systému přihlásí čtenář, může provádět následující akce

- Prohlížet všechny knihy v knihovně.
- Půjčit si vybrané knihy.
- Zobrazit knihy, které má právě půjčeny.

- Vrátit libovolnou z knih, které má půjčeny.
- Prohlížet informace o svém čtenářském účtu.
- Odhlásit se ze systému.

Pokud se do systému přihlásí administrátor, dostupné jsou následující akce

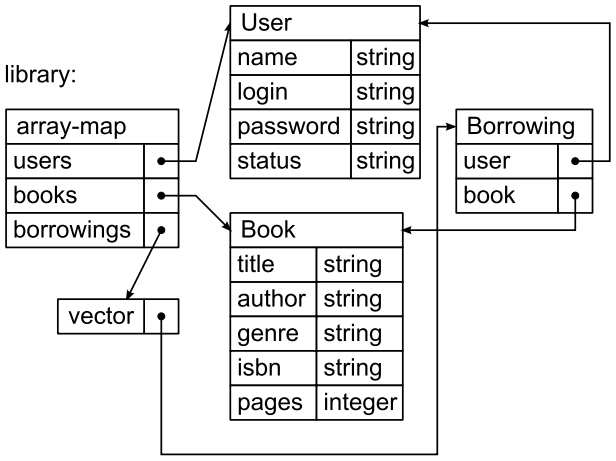
- Prohlížet všechny knihy v knihovně.
- Prohlížet všechny uživatele v knihovně.
- Přidávat či odstraňovat nové knihy.
- Přidávat či odstraňovat nové uživatele.
- Zobrazit statistiky knihovny – počet knih, uživatelů apod.
- Odhlásit se ze systému.

8.2 Implementace IS pro knihovnu

V této části je podrobně popsána implementace informačního systému knihovny. Nejprve je vytvořen doménový datový model knihovny, poté její aplikační logika a nakonec implementace akcí, které komunikují s uživatelem. Tato poslední část je pak vstupem pro generátor, který vygeneruje grafické rozhraní. Celá implementace je umístěna na přiloženém CD-ROMu, jak popisuje příloha A.

8.2.1 Datový model

Ze zadání je evidentní, že datový model musí obsahovat minimálně dvě datové struktury – jednu pro reprezentaci knihy a jednu pro reprezentaci uživatele. Další struktura pak bude použita pro reprezentaci vztahu uživatelů a knih, nebo-li takzvaných výpůjček (borrowings). Strukturu datového modelu popisuje diagram na obrázku 8.1.



Obrázek 8.1: Doménový datový model knihovny.

Aby byl tento příklad co nejjednodušší, knihy, uživatelé i výpůjčky budou uloženy jen po dobu běhu programu v operační paměti. Pozdější přidání databázové vrstvy není nikterak složitý úkol, takže je možné jej pro tuto chvíli odložit. Inicální stav knihovny, tedy příklady knih a uživatelů, se načte při startu programu z jednoduchých textových souborů.

8.2.2 Aplikační logika

Implementace aplikační logiky je snadno srozumitelná. V podstatě každá akce požadovaná ve specifikaci je implementována pomocí příslušné funkce. Funkce pracující s knihovnou jako svůj první parametr berou celou strukturu knihovny a buď vrací nějakou informaci o ní, nebo vrátí tuto strukturu nějak pozměněnou například o novou výpůjčku apod. Příkladem budiž implementace funkce `borrow-book`, která půjčí konkrétní knihu konkrétnímu uživateli a vrátí novou strukturu celé knihovny.

```
(defn borrow-book [library user book]
  (let [new-borrowings (conj (:borrowings library)
                             (Borrowing. user book))]
    (assoc library :borrowings new-borrowings)))
```

Dalším příkladem může být funkce `borrowed`, která vypíše všechny knihy půjčené určitým uživatelem

```
(defn borrowed [library user]
  (map :book (filter #(= (:user %) user) (:borrowings library))))
```

Zbytek implementace je obsažen v souboru core.clj ve složce library na přiloženém CD-ROMu.

8.2.3 Vrstva uživatelského rozhraní

Aplikační logika popsaná výše je díky funkcionálnímu přístupu k implementaci striktně bezestavová. Pro uživatelské rozhraní je však taková koncepce příliš omezující a pro uživatele nepohodlná. Uživatel jistě nechce při každé operaci pracovat s celou strukturou knihovny. Proto jsou v implementaci uživatelského rozhraní použity dva globálně přístupné atomy `*library` a `*user`, které udržují aktuální stav knihovny a aktuálně přihlášeného uživatele. Implementace akce, která půjčuje knihu, pak bere pouze jeden parametr

```
(defaction (:string (object book))
  borrow [book]
  (swap! *library #(core/borrow-book % @*user book))
  (str "The book "
       (:title book) " by " (:author book)
       " is now borrowed to "
       (:name @*user)))
```

Tato akce zároveň vrací řetězec informující uživatele o právě provedené akci. Použití direktiv pro zpřístupňování akcí ukazuje například akce `login`.

```
(defaction (:string :string :string)
  login [user pass]
  (let [usr (some #(if (= [(:login %) (:password %)] [user pass]) %)
                  (:users @*library))]
    (if usr
      (do (reset! *user usr)
          (if (= (:status usr) "admin")
            (@enable-actions-only
              (str "Admin " (:name usr) " logged in!")
```

```
            :logout :all-users :create-user
            :add-user :all-books :create-book :add-book
            :statistics)
          (@enable-actions-only
            (str "User " (:name usr) " logged in!")
            :logout :all-books :user-info :borrowed
            :borrow :return)))
    "Invalid username or password!")))
```

Posledním příkladem je anotace definující typ `(object book)`.

```
(defobject book [title :string
                 author :string
                 genre :string
                 isbn :string
                 pages :int])
```

Zbytek implementace je obsažen v souboru ui.clj v adresáři library na přiloženém CD-ROMu.

8.2.4 Generování GUI

Nyní lze přikročit k vygenerování grafického rozhraní. Předpokládejme, že s generátorem pracujeme na platformě Windows, kde je nainstalována Java verze alespoň 1.6. Předpokládejme dále, že celý obsah přiloženého CD byl zkopírován na lokální disk do adresáře, ve kterém jsou následně spouštěny všechny následující příkazy. Na ostatních platformách, na kterých běží jazyky Clojure a Java, je postup analogický.

Nejprve je třeba spustit samotný generátor. To se provede příkazem

```
java -cp "lib\*" clojure.main
```

Tím se spustí REPL pro jazyk Clojure. Zda pak stačí zadat příkazy

```
(use 'clofsui.generator 'seesaw.chooser)
(generate (choose-file))
```

Spuštěním těchto příkazů se zobrazí dialog pro výběr souboru. Zde pak stačí vybrat soubor ui.clj ve složce library, který implementuje vrstvu uživatelského rozhraní popsanou v předchozí části. V této složce by se měl objevit vygenerovaný soubor app.clj[1].

Nyní je třeba ukončit běžící instanci REPLu například stisknutím kombinace Ctrl+C nebo zadáním příkazu `(System/exit 0)`. Nyní už je aplikace připravena ke spuštění. Nejdřív je nutné spustit nový REPL, která bude znát cestu k modulům knihovny. To se provede příkazem

```
java -cp "lib\*;." clojure.main
```

Poté stačí zadat

```
(use 'library.app)
(-main)
```

Tímto se spustí vygenerované grafické rozhraní pro informační systém knihovny.

[1]Na přiloženém CD je již tento soubor obsažen. K otestování fungování generátoru je tedy vhodné tuto předgenerovanou verzi z adresáře library smazat.

Kapitola 9

Závěr

Tato práce se zabývala problémem automatického generování grafického uživatelského rozhraní z anotovaného zdrojového kódu ve funkcionálním jazyce. Jejím výsledkem je jednak analýza současných možností v oblasti automatického generování GUI a jednak návrh systému pro automatické generování GUI pro jazyk Clojure – takzvané funkcionálně strukturované uživatelské rozhraní (FSUI). Tento systém byl implementován jako framework pro jazyk Clojure a jeho možnosti byly demonstrovány na případové studii.

9.1 Zhodnocení

Cíle uvedené v úvodu práce byly splněny níže popsaným způsobem.

Provést analýzu současného stavu v oblasti automatického případně poloautomatického generování grafického uživatelského rozhraní.

Analýze současných možností se věnuje kapitola 3 a zabývá se jednak v praxi používanými frameworky pro generování GUI a jednak alternativním způsobem generování GUI pro jednodušší výpočetní modely, jako je přepisování výrazů. Během analýzy se ukázalo, že praktické frameworky jsou založeny především na OOP a grafické rozhraní generují z anotovaného doménového modelu. Naopak přepisování výrazů je příliš jednoduchý výpočetní model na to, aby byl prakticky použitelný.

Navrhnout framework podporující automatické či poloautomatické generování abstraktního GUI pro funkcionální datové struktury.

Jelikož ambicí této práce je generovat GUI bez použití doménového modelu, zabývá se kapitola 6 možnostmi generování grafického rozhraní přímo analýzou anotovaného zdrojového kódu. Z tohoto rozboru pak vychází myšlenka funkcionálně strukturovaného uživatelského rozhraní (FSUI). V téže kapitole je pak specifikováno fungování jak tohoto rozhraní tak jeho generátoru.

Vytvořit transformaci z abstraktního popisu GUI do některého běžně používaného GUI frameworku.

V kapitole 7 je implementován jak generátor tak samotné FSUI v jazyce Clojure s využitím knihovny Seesaw pro Swing GUI framework. Výsledný framework má ve srovnání s ostatními zmíněnými v analýze v kapitole 3 především tyto zásadní charakteristiky.

1. Systém metadat pro anotování kódu byl vytvářen s důrazem na maximální jednoduchost a srozumitelnost.

2. Anotace a direktivy pro grafické rozhraní minimálně interferují se zdrojovým kódem a tak ho činí čitelnějším. Nejsou však od kódu zcela odděleny, čímž snižují riziko, že při úpravách kódu se zapomene na jejich aktualizaci.

3. Generátor grafického rozhraní byl implementován s důrazem na srozumitelnost a snadnou rozšiřitelnost.

4. Typový systém byl vytvářen s cílem snadné rozšiřitelnosti a přímočaré uživatelské použitelnosti.

5. Výstup generátoru je koncipován tak, aby nebylo těžké implementovat jiné grafické klienty, kteří s ním pracují, jako jsou například webová rozhraní či mobilní aplikace.

Implementovat případovou studii demonstrující výsledky práce.

Případová studie v kapitole 8 popisuje implementaci informačního systému pro správu knihovny a tak demonstruje použití všech nástrojů, které byly v této práci navrženy a implementovány.

9.2 Náměty na rozšíření a další výzkum

Tato práce otvírá prostor pro další výzkum v oblasti automatického generování GUI. Existuje také mnoho způsobů jak implementaci zde vytvořeného frameworku rozšířit. Mezi ně patří například

- Rozšířit implementaci generátoru tak, aby umožňoval generovat výstup z více souborů, případně prohledával do hloubky moduly, na kterých vstupní modul závisí.
- Rozšířit systém metadat tak, aby umožňoval větší kontrolu nad vzhledem výsledného rozhraní. Bylo by možné například specifikovat konkrétní umístění pro některé nosiče či spouštěče, řadit akce do heirarchií a vytvářet tak různá menu apod.
- Rozšířit typový systém o typové proměnné, případně o dědičnost objektů.
- Implementovat jiného grafického klienta, například pro web či mobilní operační systémy, jako je Android.

Co se týče dalšího výzkumu, je možné například

- Zaměřit se na jiné funkcionální jazyky, než je Clojure.
- Pokusit se pokračovat směrem nastíněným v úvodu kapitoly 6 a v práci [9] a místo přepisování výrazů použít nějaký běžnější programovací jazyk.

Seznam tabulek

Seznam obrázků

Příloha A

Obsah přiloženého CD

Nosič CD přiložený k této práci obsahuje zdrojové kódy frameworku implementovaného v této práci uložené ve složce sources. Složka lib pak obsahuje jeho zkompilovanou verzi (clofsui.jar) spolu se všemi knihovnami, které jsou ke spuštění potřeba. Mezi tyto knihovny patří samotný jazyk Clojure a knihovna Seesaw pro práci s grafickým rozhraním. Ve složce library je implementace informačního systému pro správu knihovny, který je použit v případové studii. V kořenovém adresáři se pak nacházejí textové soubory s příklady dat pro knihovnický informační systém a samotný text této práce ve formátu PDF. Adresářová struktura přiloženého CD vypadá takto:

```
diplomova-prace.pdf      books.txt      users.txt

lib                      library        sources
 |- clofsui.jar           |- app.clj     |- controller
 |- clojure-1.3.0.jar     |- core.clj    |   |- actions.clj
 |- filters-2.0.235.jar   |- ui.clj      |   |- core.clj
 |- forms-1.2.1.jar                     |   |- holders.clj
 |- j18n-1.0.0.jar                      |   |- invokers.clj
 |- miglayout-3.7.4.jar                 |- directives.clj
 |- seesaw-1.3.0.jar                    |- generator.clj
 |- swing-worker-1.1.jar                |- types.clj
 |- swingx-1.6.1.jar                    |- uimodel.clj
```

Literatura

[1] SOMMERVILLE, Ian. *Software engineering*. 9. vyd. Boston: Pearson, 2011, 773 s. ISBN 978-013-7053-469.

[2] O'BRIEN, James A. a George M. MARAKAS. *Management information systems*. 9th ed. Boston: McGraw-Hill Irwin, 2009, 659 s. ISBN 00-733-7676-0.

[3] CZARNECKI, Krzysztof. *Generative programming: methods, tools, and applications*. Boston: Addison-Wesley, 2000, 832 s. ISBN 02-013-0977-7.

[4] STAHL, Thomas. *Model-driven software development: technology, engineering, management*. Hoboken: John Wiley and Sons, 2006, 428 s. ISBN 04-700-2570-0.

[5] FRANKEL, David S. *Model driven architecture: applying MDA to enterprise computing*. Indianapolis: Wiley, 2003, 328 s. ISBN 04-713-1920-1.

[6] MARTIN, Robert C. *Agile software development, Principles, Patterns, and Practices*. 1. vyd. New Jersey: Prentice-Hall, 2003, 529 s. ISBN 01-359-7444-5.

[7] TORRES, R. *Practitioner's handbook for user interface design and development*. Upper Saddle River, NJ: Prentice Hall PTR, 2002, 375 s. ISBN 978-0130912961.

[8] MACLENNAN, Bruce J. *Functional programming: practice and theory*. Reading, Mass.: Addison-Wesley, 1990, 596 s. ISBN 02-011-3744-5.

[9] JELÍNEK, Josef a Pavel SLAVÍK. *GUI Generation from Annotated Source*

Code. Proceedings of the 3rd annual conference on Task models and diagrams, s. 129–136. ACM. New York, NY, USA. 2004.

[10] *AJAX Java Framework for Rapid Application Development: OpenXava* [online]. [cit. 2012-04-05]. Dostupné z: `http://www.openxava.org`

[11] *Naked Objects*. [online]. [cit. 2012-04-05]. Dostupné z: `http://nakedobjects.codeplex.com`

[12] *Roma Framework: The new way to conceive Web Applications* [online]. [cit. 2012-04-05]. Dostupné z: `http://www.romaframework.org`

[13] The Magritte Meta-Model. *Google Project Hosting* [online]. [cit. 2012-04-05]. Dostupné z: `http://code.google.com/p/magritte-metamodel`

[14] *Trails Framework* [online]. [cit. 2012-04-05]. Dostupné z: `http://www.trailsframework.org`

[15] *JMatter* [online]. [cit. 2012-04-05]. Dostupné z: `http://jmatter.org`

[16] *Apache Isis: Domain Driven Applications, Quickly* [online]. [cit. 2012-04-05]. Dostupné z: `http://incubator.apache.org/isis`

[17] NIPKOW, Tobias a Franz BAADER. *Term rewriting and all that*. 1st pbk. ed. Cambridge, U.K: Cambridge University Press, 1999. ISBN 05-217-7920-0.

[18] *XSL Transformations (XSLT)* [online]. [cit. 2012-04-08]. Dostupné z: `http://www.w3.org/TR/xslt`

[19] ABELSON, Harold, Gerald Jay SUSSMAN a Julie SUSSMAN. *Structure and interpretation of computer programs*. 2. vyd. New York: McGraw-Hill, 1996, 657 s. ISBN 00-700-0484-6.

[20] STEELE, Guy L. *COMMON LISP: the language*. 2. vyd. Bedford, Mass.: Digital Press, 1990, 1029 s. ISBN 01-315-2414-3.

[21] DYBVIG, R. *The Scheme programming language*. 4. vyd. Cambridge, Mass.: MIT Press, 2009, 491 s. ISBN 02-625-1298-X.

[22] COAD, P. *Object-oriented programming*. New Jersey: Prentice-Hall, 1993,

582 s. ISBN 01-303-2616-X.

[23] BEN-ARI M. *Objects Never? Well, Hardly Ever!* In: *Communications of the ACM*. vol. 53, no. 09, s. 32-35, 2010

[24] GABRIEL, Richard P. Objects have failed: Notes for a Debate. In: *Dreamsongs: Blending Art & Science* [online]. [cit. 2012-04-05]. Dostupné z: `http://www.dreamsongs.com/NewFiles/ObjectsHaveFailed.pdf`

[25] CARDELLI, L. Bad engineering properties of object-oriented languages. *ACM Computing Surveys*. roč. 28, 4es, 150-es. ISSN 03600300. DOI: 10.1145/242224.242415.

[26] *Clojure – Home* [online]. [cit. 2012-04-05].
Dostupné z: `http://clojure.org`

[27] Overview. In: *Clojure v1.3 API documentation* [online]. [cit. 2012-04-04]. Dostupné z: `http://clojure.github.com/clojure`

[28] VANDERHART, Luke a Stuart SIERRA. *Practical Clojure*. New York: Springer-Verlag, 2010, 210 s. ISBN 14-302-7230-9.

[29] FOGUS, Michael a Chris HOUSER. *The joy of Clojure*. Greenwich: Manning, 2011, 328 s. ISBN 978-1-935182-64-1.

[30] RATHORE, Amit. *Clojure in action*. Shelter Island, NY: Manning, 2012, 410 s. ISBN 19-351-8259-5.

[31] SHAVIT, Nir a Dan TOUITOU. Software transactional memory. *Distributed Computing*. 1997-2-12, roč. 10, č. 2, s. 99-116. ISSN 0178-2770. DOI: 10.1007/s004460050028.

[32] *The human-computer interaction handbook*: *fundamentals, evolving technologies, and emerging applications*. 2nd ed. Editor Andrew Sears, Julie A Jacko. New York: Lawrence Erlbaum Associates, 2008, 1358 s. Human factors and ergonomics. ISBN 978-0-8058-5870-9 (VáZ.).

[33] AHO, Alfred V., John E. HOPCROFT a Jeffrey D. ULLMAN. *Data structures and algorithms*. Reading, Mass.: Addison-Wesley, 1983, 427 s. ISBN 02-010-0023-7.

[34] AKERS R. L. *Strong Static Type Checking for Functional Common Lisp*, Doctoral Dissertation, University of Texas at Austin, USA 1995.

[35] REENSKAUG, Trygve. *Models-views-controllers.* Technical note, Xerox PARC. 1979

[36] *Seesaw* [online]. [cit. 2012-04-04].
Dostupné z: `https://github.com/daveray/seesaw`

[37] LOY, Marc a Robert ECKSTEIN. *Java Swing.* 2nd ed. Sebastopol, CA: O'Reilly, 2003, 1252 s. ISBN 05-960-0408-7.